Università degli Studi di Roma "Tor Vergata"
Dipartimento di Informatica, Sistemi e Produzione

XVIII Ciclo del Corso di Dottorato di Ricerca in
"Informatica ed Ingegneria dell'Automazione"

Software Tools for Sparse Linear Algebra Computations

Alfredo Buttari

Docente guida:
Prof. Salvatore Tucci
Co-Relatori:
Ing. Salvatore Filippone
Dr. Victor Eijkhout

Coordinatore:
Prof. Daniel Pierre Bovet

May 24, 2006

to my mother, my father,
Daniele and Federica.

Contents

Contents

List of Figures

List of Figures

List of Tables

List of Algorithms

Acknowledgments

First of all I thank Salvatore Filippone for being an inexhaustible source of teaching and an enthusiastic and encouraging advisor. The way he stimulated my young researcher spirit is priceless.

I also thank all the other members of the CE Group at the "Tor Vergata" University of Rome. I expecially thank Professor Salvatore Tucci, who gave me the chance to carry out this PhD course, and Emiliano and Valeria for the priceless support and suggestions they gave me.

I wish to thank Jack Dongarra for having given me the chance to be a visiting member of the ICL Group at the University of Tennessee (UTK). Among all the people at ICL I expecially thank Victor and Julien who still represent, to me, the model of what a good researcher should be.

Among the other people at DISP (Dipartimento Informatica Sistemi e Produzione) I wish to thank my buddies Roberto and Mauro for having been admirable colleagues and good friends.

I thank Federica. Her love pushed me through many difficult moments. I'm glad to share with her my happiness for having achieved my objectives.

I owe the deepest gratitude to mom, dad and Daniele. Probably they have no idea of the "black magic" that is going to be discussed in the following pages but at least half of this thesis has been written thanks to the love and support they gave me.

Introduction

Numerical simulations are an important tool to provide detailed insight into the behavior of complex systems and to study phenomena that would be too expensive, dangerous or even impossible to analyze by direct experimentation. The quest for higher levels of detail and realism in such simulations requires enormous computational capacity and the development of *ad hoc* knowledge to utilize it in an efficient way.

Computational Science and Engineering (CSE) [66, 67, 68, 82] is a rapidly growing multidisciplinary area with connections to science, engineering, mathematics and computer science that provides to the scientist the tools and computational environment to allow the fruitful exploitation of available resources without having to resort to non-physical approximations simply to reduce the model to a tractable form. Although it includes elements from computer science, applied mathematics, engineering and science (as illustrated in figure 1 extracted from [82]), CSE focuses on the integration of knowledge and methodologies from all of these disciplines, and as such is a subject which is distinct from any of them.

The following formal definition for CSE, extracted from [66], can be given:

> *Computational Science (and Engineering) deals with the development of models and applications, algorithms for solving issues arising in the modeling process, and the matching of algorithms to architectures.*

This thesis aims at describing the work that has been done in the PhD course developed in the context of *Computational Sparse Linear Algebra* (CSLA), which is a subset of the topics that define CSE.

Many problems arising from different scientific disciplines such as fluid dynamics, astronomy, chemistry or even econometry can be modeled through the use of equations that involve partial derivatives of unknown functions. Such equations are called Partial Differential Equations (PDEs). The idea is to describe a function indirectly by a relation between itself and its partial derivatives, rather than writing down a function explicitly. The relation should be local - it should connect the function and its derivatives in the same

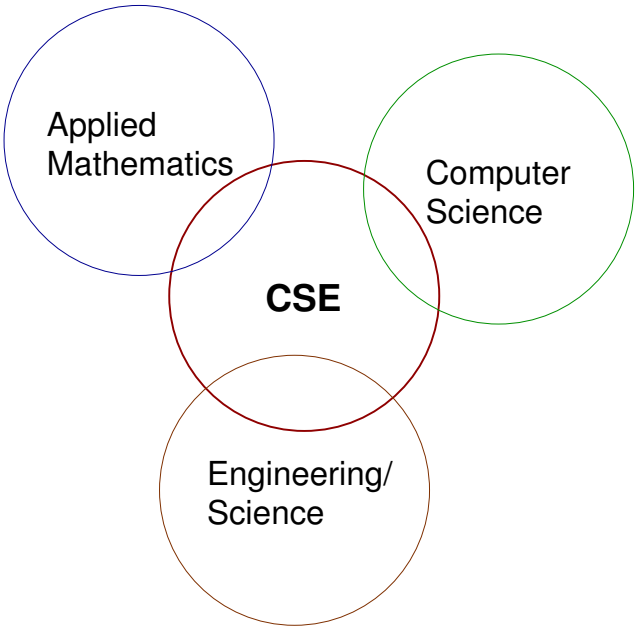

Figure 1: Graphical representation of CSE playground.

point. A solution of the equation is any function satisfying this relation. A rather common PDE is, for example, the *Poisson* equation:

$$u_{xx} + u_{yy} + u_{zz} = f \tag{1}$$

also commonly written as:

$$\Delta u = f \tag{2}$$

where $f(x, y, z)$ is a given function and Δ is the so-called *Laplacian operator*. The solution of this equation can be used to describe potential of gravitational fields in the presence of masses. The typical way to solve such equations is to approximate them by equations that involve a finite number of unknowns. This method is called *discretization*. There are several different ways to discretize a partial differential equation [64]. The simplest method uses *finite difference* approximations for the partial differential operators. The *finite element method* replaces the original function by a function which has some degree of smoothness over the global domain, but which is piecewise polynomial on simple cells, such as small triangles or rectangles. In between these two methods, there are a few conservative schemes called *finite volume methods*, which attempt to incorporate continuous conservation laws of physics.

Systems that arise from the discretization of a PDE are generally large and sparse, i.e. they have very few nonzero entries. This peculiar characteristic

come from the fact that differential operators are local operators. For example take the Poisson problem in 3 and 4 where Ω is the rectangle $(0, l_1) \times (0, l_2)$ and Γ its boundary;

$$-\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}\right) = f \quad in \quad \Omega \tag{3}$$

$$u = 0 \quad on \quad \Gamma \tag{4}$$

the finite elements discretization on a 3×5 2-d grid with square cells (see figure 2) yields the sparse matrix represented in figure 3.

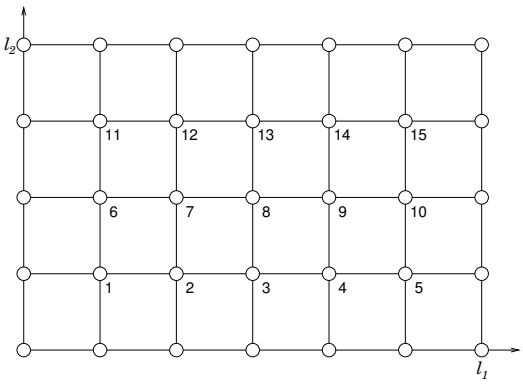

Figure 2: Sample 2-D discretization grid with square cells.

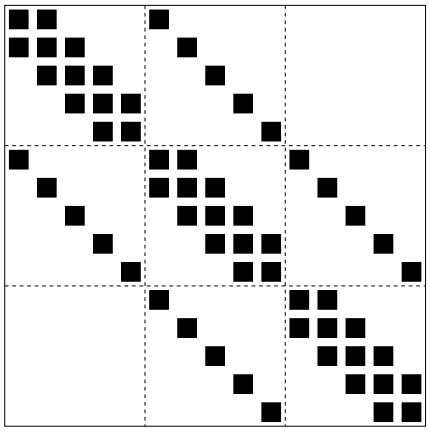

Figure 3: Sparse matrix coming from the Poisson equation discretized over a 2-D grid with square cells.

As described above, a sparse matrix is defined somewhat vaguely as a matrix which has very few nonzero elements. From a computational point of view, a matrix can be called sparse whenever special techniques can be utilized to take advantage of the large number of zero elements. When storing and manipulating sparse matrices on the computer, it is possible to modify the standard algorithms and take advantage of the sparse structure of the matrix. Sparse data is by its nature easily compressed, which can yield enormous savings in memory usage, and more importantly, even if the definition of huge depends on the hardware and the computer programs available, manipulating huge sparse matrices with the standard algorithms may be impossible due to their sheer size. These sparse techniques arise from the idea that zero elements need not be stored. Thus, traditional storage schemes used to represent dense matrices, i.e. 2-D arrays, could be replaced by more complex storage formats where only nonzero elements are stored together with their column/row index informations. At the moment there

is still no well-accepted standard on these storage formats, though some are very common and widely used such as the Compressed Sparse Row (CSR) and some others that will be presented in chapter 3.

A number of software packages exist that provide implementations for many algebraic operations that can be executed on sparse matrices like the PETSc [5] and Trilinos[46] packages or the PSBLAS[39] package, described in section 2.1, that has been the workbench for the work presented in this thesis.

The definition of Computational Science and Engineering depicted in figure 1 helps us to outline a logical partition of the topics presented in this thesis:

- **Computer Science**: chapter 2 describes the PSBLAS (Parallel Sparse Basic Linear Algebra Subroutines))software package (version 2.0). This software package has been the testbed for all of the work presented in this thesis.

  Chapter 3 describes a self-adapting software solution to implement high performance sparse algebric kernels. This technique is based on the definition of a parametric storage format for sparse matrices. According to the hardware and the input data characteristics, suitable values are chosen for the storage format parameters that provide as high as possible performance results.

- **Applied Mathematics**: chapter 1 presents a brief description of the theory of Stationary and Non-Stationary iterative solvers, namely those of the *Krylov*-subspace family, along with preconditioning methods.

  Chapter 4 introduces *Domain Decomposition Methods* theories, describes how they can be exploited to develop efficient *Additive Schwarz* preconditioning techniques, shows how they can be implemented and used in real world applications also presenting experimental results of their applications..

  Chapter 5 introduces *Multi-Grid Methods* and describes how Two-Level preconditioners can be implemented and used in real world applications providing experimental results to demonstrate the effectiveness of such methods.

- **Engineering/Science**: chapter 6 shows how CSE tools (namely the PSBLAS software package) can be used to solve problems arising from the modeling of engineering/science problems. Section 6.2 describes the integratio of the PSBLAS solvers in an application developed for combustion engines fluid dynamics studies. The mathematical model discussed in chapter 6.2 is the complete system of general *unsteady*

Navier-Stokes equations, coupled with chemical kinetic and spray droplet dynamic models.

Introduction

Part I

Sparse Linear Algebra and Iterative solvers

Chapter 1

Iterative Solvers

Contents

Iterative methods for solving general, large sparse linear systems have been gaining popularity in many areas of scientific computing. Until around the '80s direct solution methods where often preferred to iterative methods due to their robustness and predictable behavior. However a number of efficient iterative solvers were discovered and the increased need for solving very large linear systems triggered a noticeable and rapid shift toward iterative techniques in many applications. This trend can be traced back to the 1960s and 1970s when two important developments revolutionized solution methods for sparse linear systems. First was the realization that one can take advantage of "sparsity" to develop special methods and implement operations in a way that can be quite economical when compared to the dense equivalent. Second was the application of preconditioning techniques to conjugate gradient-like methods for solving linear systems. It was found that adding preconditioners to *Krylov* subspace iterative methods could provide efficient and simple general-purpose procedures that could compete with

direct solvers. The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix which is the set of its eigenvalues. Preconditioning involves a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a *preconditioner*. The use of preconditioning techniques lies on the idea of exploiting concepts from direct solvers theory. A good preconditioner improves the convergence of the iterative method sufficiently to overcome the extra cost of constructing and applying it.

Nowadays three dimensional models are commonplace and iterative methods are almost mandatory. The memory and the computational requirements for solving three dimensional PDEs, or two dimensional ones involving many degrees of freedom per point, may seriously challenge even the most efficient direct solvers available today.

The term "iterative method" refers to a wide range of techniques that use successive approximations to obtain more accurate solution to a linear system at each step, starting from an initial guess. A high-level definition of an iterative method involves describing

- an initial guess x_0
- an iteration function $x_k = f(A, b, x_{k-1})$ where A and b are, respectively, the system matrix and right-hand side
- a stopping criterion that defines the convergence of the method.

The most commonly used iterative methods can be classified in two different categories:

Stationary methods: these methods are older, simpler to understand and implement, but usually not as effective. This class includes the well known *Jacobi* method or the *Gauss Seidel* and *SOR* ones.

Nonstationary methods: nonstationary methods are a relatively recent development; their analysis is usually harder to understand but they can be highly effective. The most famous ones are based on the idea of sequences of orthogonal vectors. This class includes the commonly used *Conjugate Gradient (CG)*, *BiConjugate Gradient (BiCG)* or *Generalized Minimal Residual (GMRES)* methods.

The description in this chapter follows closely that in [7] and [64].

1.1 Stationary Methods

Iterative methods that can be expressed in the simple form

$$x^{(k)} = Bx^{(k-1)} + c \tag{1.1}$$

(where neither B nor c depend upon the iteration count k) are called stationary iterative methods. Stationary methods are the first iterative methods used for solving linear systems and are based on the *relaxation of the coordinates*. Beginning with a given approximate solution, these methods modify the components of the approximation, one or a few at a time and in a certain order, until convergence is reached. Each of these modifications is called a *relaxation step* and it is aimed at annihilating one or few components of the residual vector.

Given an $n \times n$ real[1] matrix A and a real n-vector b the problem considered is:

$$Ax = b \tag{1.2}$$

The formula 1.2 is a linear system where A is the *coefficient matrix*, b is the *right-hand side* vector and x the vector of *unknowns*. Each of the equations in 1.2 can be expressed as

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \tag{1.3}$$

Solving 1.3 for x_i while assuming the other entries of x remain fixed, yields:

$$x_i = (b_i - \sum_{j \neq i} a_{i,j} x_j)/a_{ii} \tag{1.4}$$

This suggests an iterative method defined by:

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j} x_j^{(k-1)})/a_{ii} \tag{1.5}$$

The step in 1.5 is a component-wise form of an iteration of the Jacobi method. The order in which the equation are examined is irrelevant since the Jacobi method treats them independently. For this reason the Jacobi method is also known as the *method of simultaneous displacements*.

In matrix terms, the definition of the Jacobi method in 1.5 can be expressed as:

$$x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b \tag{1.6}$$

[1] the following discussion will be developed in the domain of the real numbers $\mathbb{R}$. The presented theories are general and thus are also valid in $\mathbb{C}$

where D, $-L$ and $-U$ are the diagonal, the strictly-lower and the strictly-upper part of A. Comparing equation 1.6 with equation 1.1 it's obvious why the Jacobi method is a stationary one.

The *Gauss Seidel* method can be easily derived from the Jacobi one under the assumption that previously computed results can be used as soon as they are available. Formally the step in 1.5 becomes:

$$x_i^{(k)} = (b_i - \sum_{j<i} a_{i,j} x_j^{(k)} - \sum_{j>i} a_{i,j} x_j^{(k-1)})/a_{ii} \tag{1.7}$$

which is the component-wise form of a step of the Gauss Seidel method. At each k-th step the component $x_i^{(k)}$ of the vector of unknowns depends upon all the $x_1^{(k)}...x_{i-1}^{(k)}$ and thus all the updates cannot be performed at the same time, as in the Jacobi method, but must be serialized. Moreover the iterate $x^{(k)}$ depends on the order in which equations are examined, this is why the Gauss Seidel method is also called the *method of successive displacements*.

The definition of the Gauss Seidel method in matrix terms is:

$$x^{(k)} = (D - L)^{-1}(Ux^{(k-1)} + b) \tag{1.8}$$

where D, L and U have the same meaning as in equation 1.6.

It is easily possible to show that if a stationary method converges, then the limit is a solution of the original system. All the stationary iterative methods define a sequence of iterates as defined in equation 1.1 where B is a certain *iteration matrix*. If the iteration in 1.1 converges, then its limit x satisfies:

$$x = Bx + c \tag{1.9}$$

Whereas the matrix A can be split like $A = D - L - U$ the previous becomes:

$$x = D^{-1}(L - U)x + D^{-1}b \tag{1.10}$$

that is equivalent to equation 1.2.

Thanks to the simplicity of these stationary methods, several results and laws have been found that state when and how fast such methods converge. The following theorem states that a stationary method converges if and only if the spectral radius of the matrix in 1.1 is less than one.

Theorem 1 *Let B be a square matrix such that $\rho(B) < 1$. Then $I - B$ is nonsingular and the iteration 1.1 converges for any c and x_0. Conversely if the iteration 1.1 converges for any c and x_0, then $\rho(B) < 1$.*

Since it is expensive to compute the spectral radius of a matrix, sufficient conditions that guarantee convergence can be useful in practice. One such sufficient condition can be obtained by utilizing the inequality $\rho(B) \leq \|B\|$ for any matrix norm.

Corollary 1 *Let B be a square matrix such that $\|B\| < 1$ for some matrix norm $\|\cdot\|$. Then $I - B$ is nonsingular and the iteration 1.1 converges for any initial vector x_0.*

Apart from knowing that the sequence 1.1 converges, it is also desirable to know how fast it converges. The error $d_k = x_k - x$ at step k satisfies:

$$d_k = B^k d_0$$

The quantity:

$$\rho = \lim_{k\to\infty} \left(\frac{\|d_k\|}{\|d_0\|} \right)^{1/k}$$

is called the *convergence factor* of the sequence 1.1. It can be proven that $\rho = \rho(B)$. The *convergence rate* τ is the natural logarithm of the inverse of the convergence factor:

$$\tau = -ln\rho$$

Unfortunately there are no such useful properties for non stationary iterative methods.

1.2 Nonstationary Methods

The purpose of this section is to give a brief introduction on Krylov subspace nonstationary iterative methods that are probably the most commonly used techniques for the solution of sparse linear systems. These methods are based on *projection processes* onto Krylov subspaces.

Consider again the linear system:

$$Ax = b \tag{1.11}$$

where A is a $n \times n$ real matrix. Projection techniques extract approximate solutions of the above problem from subspaces of $\mathbb{R}^n$. If $\mathcal{K}$ is this subspace of *candidate approximations*, or *search subspace*, an m is its dimension, then m constraints must be imposed to be able to extract such an approximation. A common choice is to impose m orthogonality conditions. Specifically the residual vector $b - Ax$ is constrained to be orthogonal to m linearly independent vectors. This defines another subspace $\mathcal{L}$ called *subspace of constraints*. This basic theory, common to many iterative solvers, is usually referred as *Petrov-Galerkin conditions*. There are two broad classed of projection methods:

- *orthogonal*: the subspace $\mathcal{L}$ is the same as $\mathcal{K}$.

- *oblique*: the subspace $\mathcal{L}$ is not the same as $\mathcal{K}$ and may be completely unrelated to it.

Thus a projection technique onto the subspace $\mathcal{K}$ and orthogonal to $\mathcal{L}$ is a process which finds an approximate solution $\tilde{x}$ to 1.11 by imposing the conditions that $\tilde{x}$ belongs to $\mathcal{K}$ and the new residual vector $b - Ax$ be orthogonal to $\mathcal{L}$,

$$\text{Find} \quad \tilde{x} \in \mathcal{K}, \qquad \text{such} \quad \text{that} \quad b - A\tilde{x} \perp \mathcal{L} \tag{1.12}$$

To better exploit the knowledge of an initial guess x_0 to the solution, the approximation must be sought in the affine space $x_0 + \mathcal{K}$ instead of the homogeneous vector space $\mathcal{K}$. The approximate problem should be redefined as:

$$\text{Find} \quad \tilde{x} \in x_0 + \mathcal{K}, \qquad \text{such} \quad \text{that} \quad b - A\tilde{x} \perp \mathcal{L} \tag{1.13}$$

If $\tilde{x}$ is written in the form $\tilde{x} = x_0 + \delta$, and the initial residual vector is $r_0 = b - Ax_0$, the previous equation becomes:

$$\text{Find} \quad \tilde{x} \in x_0 + \mathcal{K}, \qquad \text{such} \quad \text{that} \quad r_0 - A\delta \perp \mathcal{L} \tag{1.14}$$

Finally the approximate solution can be defined as

$$\begin{aligned} \tilde{x} &= x_0 + \delta, \quad \delta \in \mathcal{K}, \\ (r_0 - A\delta, \omega) &= 0, \quad \forall \omega \in \mathcal{L} \end{aligned}$$

Figure 1.1 sketches a visual interpretation of the orthogonality conditions imposed to the new residual vector.

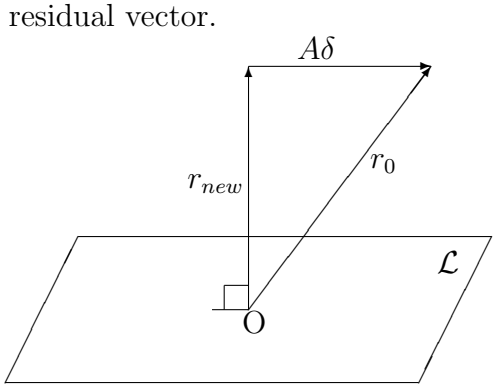

Figure 1.1: Interpretation of orthogonality conditions.

Usually iterative methods consist of consecutive repetition of this projection step. At each step k, x_0 and r_0 are assumed to be respectively the approximate solution and the related residual found at step $k - 1$.

Let $V = [v_1, ..., v_m]$ an $n \times m$ matrix whose columns form a basis of $\mathcal{K}$ and $W = [w_1, ..., w_m]$ an $n \times m$ matrix whose columns form a basis of $\mathcal{L}$. Then the approximate solution can be written as a linear combination of the vectors in V:

$$\tilde{x} = x_0 + Vy$$

and the orthogonality conditions:

$$W^T AVy = W^T r_0$$

Combining the above equations, a matrix formulation of a projection step is obtained:

$$\tilde{x} = x_0 + V(W^T AV)^{-1} W^T r_0 \tag{1.15}$$

It is important to note that equation 1.15 is only valid when $W^T AV$ is non-singular (and thus invertible).

Algorithm 1 Projection method prototype

```
k := 1
repeat
    Select a pair of subspaces K and L
    Choose bases V = [v_1, ..., v_m] and W = [w_1, ..., w_m] for K and L
    r_{k-1} := b - A x_{k-1}
    y_k := (W^T A V)^{-1} W^T r_{k-1}
    x_k := x_{k-1} + W y_k
    k := k + 1
until Convergence
```

Most of the iterative solvers used nowadays are based on projection techniques onto Krylov subspaces which are subspaces spanned by vectors of the form $p(A)v$ where p is a polynomial.

A Krylov subspace method can be formally defined as a method for which the subspace $\mathcal{K}_m \in \mathbb{R}^m$ is the Krylov subspace:

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

Different flavors of Krylov subspace methods arise from different choices of the subspace $\mathcal{L}_m$ and from the way in which the system is *preconditioned* (see 1.3). Although all the techniques provide the same type of polynomial approximations, the choice of $\mathcal{L}_m$ usually has an important impact on the efficiency and quality of the iterative method with respect to several algebraic properties of the system to be solved. Four broad choices for $\mathcal{L}_m$ give rise to the best known techniques:

1. *Ritz-Galerkin* approach: $\mathcal{L}_m = \mathcal{K}_m$ and thus requires that $b - Ax_k \perp \mathcal{K}_k(A, r_0)$. This is the choice for the very well known *Conjugate Gradient* method (see section 1.2.2).

2. *minimum residual* approach: this is a variant of the previous approach and defines $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$. It can be proven that such a choice is equivalent to minimize the residual norm over all the vectors in $x_0 + \mathcal{K}_m$ (hence the name *minimum residual*). This approach is the base of another well known method called *Generalized Minimum Residual* or GMRES (see section 1.2.1).

3. *Petrov-Galerkin* approach: as stated before, this condition simply requires the residual to be orthogonal to some suitable$k$-dimensional space $\mathcal{L}_m$. A good choice is define the subspace $\mathcal{L}_m$ to be a Krylov subspace method associated with A^T, namely $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$. This is the basis of methods such as the *BiConjugate Gradient* one (see section 1.2.3). Recently some variants have been proposed like the *BiConjugate Gradient Stabilized* or BiCGStab method.

4. *minimum error* approach: require $\|x - x_k\|_2$ to be minimal over $A^T\mathcal{K}_k(A^T, r_0)$.

The subsections 1.2.1, 1.2.2, 1.2.3 and 1.2.4 will give some detail of preconditioned versions the most commonly used methods while section 1.3 will briefly present some preconditioning techniques. For deeply detailed dissertation about iterative methods (either stationary or nonstationary) and preconditioners please refer to [7, 28, 64].

1.2.1 Generalized Minimum Residual Method

The Generalized Minimum Residual (GMRES) method is an extension of the MINRES one (which is only applicable to symmetric systems) to unsymmetric systems. It is based on the generation of a sequence of orthonormal vector by means of the so called "Arnoldi method" which is a modified Gram-Schmidt orthogonalization applied to the Krylov sequence $\{A^k, r_0\}$:

$u_i = Av_i$
for $k = 1$ to i **do**
  $u_i = u_i - (u_i, v_k)v_k$
end for
$v_{i+1} = u_i/\|u_i\|$

The inner product coefficients (u_i, v_k) and $\{w_i\}$ are stored in an upper Hessemberg matrix $\bar{H}_i$. The GMRES iterates are constructed as

$$x_i = x_0 + Vy$$

where $V = [v_1, ..., v_i]$ is the matrix whose column are the result of previous iterates and the coefficient vector y has been chosen to minimize the residual norm $\|b - Ax_i\|$.

Unlike MINRES and CG (discussed in section1.2.2), these orthogonal vectors cannot be generated with short recurrences and thus a main practical disadvantage of GMRES is that all the successive residual vectors must be stored and that the construction of the projected system becomes increasingly complex. Thus the GMRES method becomes unpractical as the number of steps m increases due to the high memory and computational requirements. One simple solution to this problem is based on "restarts" which yields the GMRES(m) method. This technique consists of restarting the method at some step m, i.e. clearing the accumulated data and choosing intermediate result x_m as the new initial guess x_0:

Algorithm 2 Restarted GMRES prototype

1: Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|$ and $v_1 = r_0/\beta$
2: Generate the Arnoldi basis anh $\bar{H}_m$ starting with v_1
3: Compute y_m which minimizes $\|\beta e_1 - \bar{H}_m y$ and $x_m = x_0 + V_m y_m$
4: If satisfied the **Stop**, else set $x_0 := x_m$ and **GoTo** 1.

The original method with no restarting is often referred to as the "full" GMRES. The choice of m requires some skill and experience with the type of problems that one wants to solve. Taking m too small could result in poor convergence or no convergence at all.

Since the dimension of the Krylov subspace is bounded by n, the method terminates in at most n steps if rounding errors are absent. In practice this finite termination property is of no importance, since these iterative methods are attractive, with respect to direct ones, only when they deliver a suitable approximation to the solution in far less than n steps.

1.2.2 Conjugate Gradient Method

This method is derived from the symmetric Lanczos algorithm which is a simplification of the Arnoldi's method for the particular case when the matrix is symmetric. In this case the Hessemberg matrix $\bar{H}_m$ (discussed in section 1.2.1) becomes symmetric tridiagonal. This leads to a three term

recurrence in the Arnoldi process and short-term recurrences for solution algorithms such as the Conjugate Gradient one.

The Conjugate Gradient method (CG) is an effective method for symmetric positive definite systems. This method proceeds by generating vector sequences of iterates (successive approximations to the solution), residual corresponding to the iterates, and search directions used to update the iterates and the residuals. Although the length of these sequences can become very large, only a small number of vectors need to be kept in memory. In every iteration of the method two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy the previously discussed orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.

The iterates x_i are updated in each iteration by a multiple α_i of the search direction vector p_i:

$$x_i = x_{i-1} + \alpha_i p_i$$

Correspondingly the residuals $r_i = b - Ax_i$ are updated as:

$$r_i = r_{i-1} - \alpha q_i \qquad where \qquad q_i = Ap_i \tag{1.16}$$

The choice $\alpha = \alpha_i = r_{i-1}^T r_{i-1} / {p_i}^T A p_i$ minimizes ${r_i}^T A^{-1} r_i$ over all possible choices of α.

The search directions are updated using the residuals

$$p_i = r_i + \beta_{i-1} p_{i-1} \tag{1.17}$$

where the choice $\beta_i = {r_i}^T r_i / {r_{i-1}}^T r_{i-1}$ ensures that r_i and r_{i-1} are orthogonal. Algorithm 3 contains the pseudocode for the preconditioned CG method.

Choosing $M = I$, then the unpreconditioned or standard conjugate gradient method results. The unpreconditioned version constructs the ith iterate x_i as an element of $x_0 + span\{r_0, Ar_0, ..., A^{i-1}\}$ so that $(x_i - \hat{x})^T A (x_i - \hat{x})$ is minimized where $\hat{x}$ is the exact solution of equation 1.2. This minimum is guaranteed to exist in general only if A is symmetric positive definite.

Stopping criteria are often based upon the norm of the current residual vector r_i. A naive stopping criterion, for example, would stop the procedure is less than some given value eps. more robust stopping criteria are based upon estimating the error in x_i with respect to $\hat{x}$ with information obtained from the iteration parameters α_i and β_i. Anyway, in general a good stopping criteria should[2]

[2]These considerations are valid for all the iterative methods presented here and not just for the CG method.

Algorithm 3 Preconditioned Conjugate Gradient Method

```
Compute r_0 = b − Ax_0 for some initial guess x_0
for i = 1 to ... do
    solve Mz_{i−1} = r_{i−1}
    ρ_{i−1} = r_{i−1}^T z_{i−1}
    if i = 1 then
        p_1 = z_0
    else
        β_{i−1} = ρ_{i−1}/ρi − 2
    end if
    q_i = Ap_i
    α_i = ρ_{i−1}/p_i^T q_i
    x_i = x_{i−1} + α_i p_i
    r_i = r_{i−1} + α_i q_i
    check convergence; continue if necessary
end for
```

1. identify when the error $e_i \equiv x_i - \hat{x}$ is small enough to stop,
2. stop if the error is no longer decreasing or decreasing too slowly, and
3. limit the maximum amount of time spent iterating.

Accurate predictions of the convergence of iterative methods are difficult to make, but useful bounds can be often obtained. For the CG method the error can be bounded in terms of the spectral condition number κ of the matrix $M^{-1}A$. It can be proved that:

$$\|x_i - \hat{x}\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \|x_0 - \hat{x}\|_A$$

In practice the speed of convergence can be considerably faster, most notably in situations in which the extreme eigenvalues are relatively well separated from the rest of the spectrum.

1.2.3 BiConjugate Gradient Method

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences. The GMRES method retains orthogonality of the residuals by using long recurrences at the cost of a larger storage demand. The BiConjugate Gradient method (BiCG) takes another approach replacing the orthogonal sequence

of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization.

The BiConjugate Gradient can be derived from the Lanczos "biorthogonalization" in exactly the same way as the Conjugate Gradient can be derived by the Lanczos orthogonalization one. The Lanczos biorthogonalization method is an extension to nonsymmetric matrices of the symmetric Lanczos algorithm, based on biorthogonal sequences instead of orthogonal ones.

Implicitly the algorithm solves not only the original system $Ax = b$ but also a dual linear system $A^T\tilde{x} = \tilde{b}$ with A^T. This dual system is often ignored in the formulations of the algorithm.

The BiConjugate Gradient algorithm process onto

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

orthogonally to

$$\mathcal{L}_m = span\{w_1, A^T w_1, ..., (A^T)^{m-1}w_1\}$$

taking $v_1 = r_0/\|r_0\|_2$. The vector w_1 is arbitrary provided that $(v_1, w_1) \neq 0$ but it is often chosen to be equal to v_1. If there is a dual system $A^T\tilde{x} = \tilde{b}$ to solve with A^T, then w_1 is obtained by scaling the initial residual $\tilde{b} - A^T\tilde{x}$.

The update relations for residual in the Conjugate Gradient method are augmented in the BiConjugate Gradient method by relations that are similar but based in A^T instead of A.

$$r_i = r_{i-1} - \alpha_i A p_i, \qquad \tilde{r}_i = \tilde{r}_{i-1} - \alpha_i A^T \tilde{p}_i$$

$$p_i = r_{i-1} + \beta_{i-1} p_{i-1}, \qquad \tilde{p}_i = \tilde{r}_{i-1} + \beta_{i-1} A^T \tilde{p}_{i-1}$$

The choices

$$\alpha_i = \frac{\tilde{r}_{i-1}^T r_{i-1}}{\tilde{p}_i^T A p_i}, \qquad \beta_i = \frac{\tilde{r}_i^T r_i}{\tilde{r}_{i-1}^T r_{i-1}}$$

ensure the biorthogonality relations

$$\tilde{r}_i^T r_j = \tilde{p}_i^T A p_j = 0 \qquad \text{if} \quad i \neq j$$

Algorithm 4 contains the pseudocode for the BiCG method.

Few theoretical results are known about the convergence of BiCG. For symmetric positive definite systems the method delivers the same results as CG but at twice the cost per iteration. For unsymmetric matrices it has been shown that in phases of the process where there is significant reduction of the norm of the residual, the method is more or less comparable to GMRES. In practice this is often confirmed, but it also observed that the convergence behavior may be quite irregular and the method may even break down.x

Algorithm 4 Preconditioned BiConjugate Gradient Method

```
Compute r_0 = b − Ax_0 for some initial guess x_0
Choose r̃_0 (for example r̃_0 = r_0)
for i = 1 to ... do
    solve M z_{i−1} = r_{i−1}
    solve M^T z̃_{i−1} = r̃_{i−1}
    ρ_{i−1} = z_{i−1}^T z̃_{i−1}
    if ρ_{i−1} = 0 method fails
    if i = 1 then
        p_1 = z_0
        p̃_1 = z̃_0
    else
        β_{i−1} = ρ_{i−1}/ρi − 2
        p_i = z_{i−1} + β_{i−1} p_{i−1}
        p̃_i = z̃_{i−1} + β_{i−1} p̃_{i−1}
    end if
    q_i = A p_i
    q̃_i = A^T p̃_i
    α_i = ρ_{i−1}/p̃_i^T q_i
    x_i = x_{i−1} + α_i p_i
    r_i = r_{i−1} + α_i q_i
    r̃_i = r̃_{i−1} + α_i q̃_i
    check convergence; continue if necessary
end for
```

1.2.4 BiConjugate Stabilized Gradient Method

BiCGStab is based on the observation that the BiCG vector r_i is orthogonal to the entire subspace $\mathcal{K}_i(A^T, w_1)$. As a result it is possible to construct iteration methods by which x_i are generated so that $r_i = Q_i(A)P_i(A)r_0$ with other ith degree polynomials Q_i. An obvious possibility is to let Q_i be a polynomial of the form

$$Q_i(t) = (1 - \omega_{(1)}t)(1 - \omega_{(2)}t)...(1 - \omega_{(i)}t)$$

and to select suitable constants $\omega_{(j)}$. This expression leads to an almost trivial recurrence relation for the Q_i.

In BiCGStab, $\omega_{(j)}$ in the jth iteration step is chosen as to minimize r_j, with respect to $\omega_{(j)}$, for residuals that can be written as $r_j = Q_j(A)P_j(A)r_0$.

BiCGStab can be interpreted as the product of BiCG and repeatedly applied GMRES(1). At least locally a residual vector is minimized, which leads to a considerably smoother convergence behavior.

Algorithm 5 Preconditioned BiConjugate Gradient Stabilized Method

Compute $r_0 = b - Ax_0$ for some initial guess x_0
Choose $\tilde{r}_0$ (for example $\tilde{r}_0 = r_0$)
for $i = 1$ to ... **do**
  $\rho_{i-1} = \tilde{r}^T r_{i-1}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$ **then**
    $p_1 = r_0$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho i - 2)/(\alpha_{i-1}/\omega_{i-1})$
    $p_i = r_{i-1} + \beta_{i-1}(p_{i-1} - \omega_{i-1} v_{i-1}$
  end if
  solve $M\hat{p} = p_i$
  $v_i = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T v_i$
  $s = r_{i-1} - \alpha_i v_i$
  check norm of s; if small enough set $x_i = x_{i-1} + \alpha_i \hat{p}$ and stop
  solve $M\hat{s} = s$
  $t = A\hat{s}$
  $\omega_i = t^T s / t^T t$
  $x_i = x_{i-1} + \alpha_i \hat{p} + \omega_i \hat{s}$
  $r_i = s - \omega_i t$
  check convergence; continue if necessary (provided $\omega_i \neq 0$)
end for

1.3 Preconditioners

Lack of robustness is a widely recognized weakness of iterative solvers when compared to direct ones. This drawback hampers the acceptance of iterative methods in industrial applications despite their intrinsic appeal for very large linear systems. Both the efficiency and robustness of iterative solvers can be improved by the usage of *preconditioning*.

The convergence rate of an iterative method depends on the spectral properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent in the sense that it has the same solution, but that has more favorable spectral properties. A *preconditioner* is a matrix that performs such transformation. For instance, if the preconditioning matrix M approximates the coefficient matrix A in some way, the transformed system

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

has the same solution as the original system $Ax = b$ but the spectral properties of its coefficient matrix $M^{-1}A$ could make it more suitable for being solved by means of an iterative method.

The word "preconditioning" has been originally used by Turing in [74] and since then became the standard terminology for problem transformation in order to make solution easier. The first application of this word to the idea of improving the convergence of an iterative method is in [33] and in [4] preconditioning techniques are applied to the conjugate gradient method.

In general, the reliability of iterative techniques depends much more on the quality of the preconditioner than on the particular Krylov subspace method used.

The general problem of finding a preconditioner for a linear system $Ax = b$ is to find a Maire M (the preconditioner or preconditioning matrix) with the properties that:

1. M is a good approximation of A in some sense,
2. the cost of the construction of M is not too high and
3. the system $Mx = b$ is much easier to solve than the original system.

Krylov subspace methods need the operator of the linear system only for computing matrix-vector products. This means that $M^{-1}A$ need not be formed explicitly. Instead $u = M^{-1}Av$ it is formed by first computing $w = Av$ and then obtain u by solving $Ku = w$. Note that when solving the preconditioned system using a Krylov subspace method,quite different subspaces than for the original system will be obtained. The aim is that approximation in this new sequence of subspaces will approach the solution more quickly than in the original subspaces.

There are different ways of implementing preconditioning that can also lead to quite difference convergence behaviors. Three main different implementations are as follows:

Left preconditioning: apply the iterative method to $M^{-1}AX = M^{-1}b$. However the symmetry of M and A does not imply the symmetry of $M^{-1}A$ and thus the CG method (and all those methods only usable with symmetric matrices) cannot be used in a straightforward way. A simple solution to this problem is based on the observation that $M^{-1}A$ is self-adjoint for the M-inner product

$$(x, y)_M \equiv (Mx, y) = (x, My)$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M$$

Therefore an alternative is to replace the usual Euclidean inner product in the Conjugate Gradient algorithm by the M-inner product. Popular formulations of CG are based on this observation.

If using a Minimum Residual method, it should be noted that with left preconditioning the preconditioned residuals $M^{-1}(b - Ax_k)$ are being minimized which may be quite different form the residuals $b - Ax_k$. This could have consequences for stopping criteria that are based on the norm of the residual.

Right preconditioning: apply the iterative method to $AM^{-1}y = b$ with

$$x = M^{-1}y. \tag{1.18}$$

This form of preconditioning also does not lead to a symmetric product when A and M are symmetric. With right preconditioning special attention must be given to stopping criteria that are based upon the error: $\|y - y_k\|_2$ may be much smaller than the error-norm $\|x - x_k\|_2$ on which the stopping criterion is based.

Right preconditioning has the advantage that it only affects the operator and not the right-hand side. This may be useful in the design of software.

Two-sided preconditioning: for a preconditioner M with $M = M_1 M_2$ the iterative method can be applied to

$$M_1{}^{-1} A M_2{}^{-1} z = M_1{}^{-1} b \tag{1.19}$$

with $x = M_2{}^{-1} z$. The matrices M_1 and M_2 are called left and right preconditioners respectively.

If M is symmetric and $M_1 = M_2{}^T$ (i.e. M is available in the form of an incomplete Cholesky factorization $M = LL^T$) the iteration matrix also becomes symmetric, hence the CG method can be applied.

The splitting of M is in practice not needed. By rewriting the steps of the method it is usually possible to reintroduce a computational step

$$solve \quad u \quad from \quad Mu = v$$

that is a step that applies the preconditioner in its entirety.

There is also another way to apply such a preconditioning scheme that consist of applying an unpreconditioned iterative method to the system in equation 1.19:

1. Take a preconditioned iterative method and replace every occurrence of M by I.
2. Remove any vectors from the algorithm that have become duplicates in the previous step.
3. Replace every occurrence of A in the method by ${M_1}^{-1}A{M_2}^{-1}$.
4. After the calculation of the initial residual add the step

$$r_0 \leftarrow {M_1}^{-1} r_0.$$

5. At the end of the method add the step

$$x \leftarrow {M_2}^{-1} x.$$

   where x is the final calculated solution.

The choice of M varies from purely "black box" algebraic techniques which can be applied to general matrices, to "problem dependent" preconditioners which exploit special features of a particular problem class. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. Moreover there is very little theory for what one can expect a priori with a specific type of preconditioner.

It is important to note that preconditioning increases the computational complexity of an iterative method and thus the use of a preconditioner must be justified by an adequate reduction in the number of iterations.

Consider, for example, methods with a fixed amount of computational overhead per iteration step independent of the iteration number (CG, BiCG and MINRES are among those) and denote by t_A the computing time for the matrix-vector product with A and by t_O the computational overhead per iteration step. The computing time for k iteration steps is given by

$$T_U = k(t_A + t_O). \tag{1.20}$$

For the preconditioning process the following assumptions can be made with respect to computing time:

1. the action of the preconditioner, for instance the computation of $M^{-1}w$ takes αt_A.
2. the construction cost for the preconditioner is t_C.
3. preconditioning reduces the number of iterations by a factor f ($f > 1$).

The computing time for the preconditioned process to obtain an approximation comparable to the unpreconditioned process can be expressed as

$$T_P = \frac{k}{f}((\alpha + 1)t_A + t_O) + t_C. \tag{1.21}$$

The goal of preconditioning is that $T_P < T_U$, which is the case if

$$f > \frac{(\alpha + 1)t_A + t_O}{t_A + t_O - \frac{t_C}{k}}.$$

Obviously the construction of expensive preconditioners is pointless if the number of iterations k for the unpreconditioned process is low. It is thus realistic to consider only cases where k is so large that the initial cost plays no role. Furthermore, in many cases the matrix-vector products are the most expensive part of the computation ($t_A > t_O$) and thus preconditioning gives some benefit only if the factor f is significantly bigger than $\alpha + 1$.

As an example, let's compare the choice of using an incomplete factorization preconditioner (see section 1.3.2) versus an unpreconditioned solver. The following timing have been measured on the solution of the matrix *kivap2* arising from the application described in section 6.2. The solution is computed using a BiCGSTAB iterative method as implemented in the PSBLAS library (see section 2.6) on an AMD-Athlon 1800 architecture. Figure 1.2 shows the total time for the solution in the cases described.

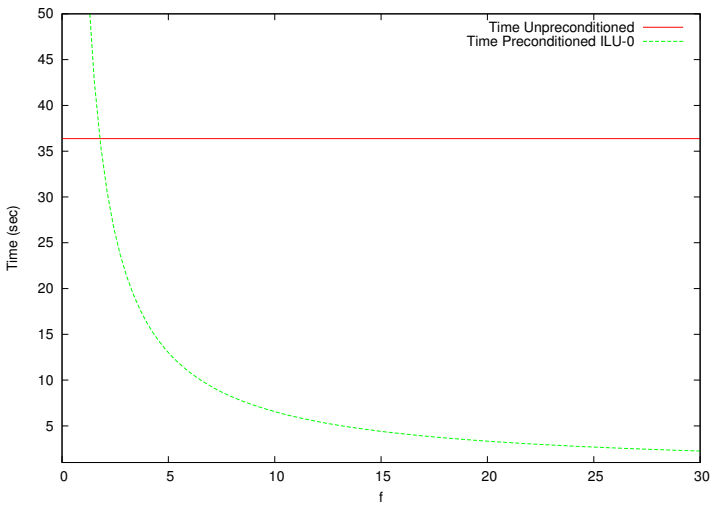

Figure 1.2: The execution times for an unpreconditioned BiCGSTAB solver and an ILU-0 preconditioned one (assume $f = 1$ in the first case).

It turns out that the total number of iteration should be reduced by at least a factor of 1.77 by the ILU preconditioner to achieve some reduction in the time spent in the solver. Fortunately in the case of our example the number of iterations has been reduced by a factor of 12 and the time for the solution is reduced from 36.41 seconds to 4.98 seconds.

For methods like GMRES the overhead cost increases quadratically with the number of iterations and thus the situation is slightly more complicated.

1.3.1 Jacobi and Block Jacobi preconditioning

The simplest preconditioner consists of just the diagonal of the matrix:

$$m_{i,j} = \begin{cases} a_{i,j} & \text{if} i = j \\ 0 & \text{otherwise} \end{cases}$$

This is known as the (point) Jacobi preconditioner.

It is possible to use this preconditioner without the need for any extra storage beyond that of the matrix itself. However, division operations are usually quite costly and then a memory area is allocated to store the inverse of the diagonal.

The block-version of the Jacobi preconditioning technique can be derived by a partitioning of the variables. If the index set $S = \{1, ..., n\}$ is partitioned as $s = \bigcup_i S_i$ with the sets S_i mutually disjoint, then

$$m_{i,j} = \begin{cases} a_{i,j} & \text{if } i \text{ and } j \text{ are in the same subset} \\ 0 & \text{otherwise} \end{cases}$$

The preconditioner is now a block-diagonal matrix.

Often natural choices for the partitioning suggest themselves:

- In problem with multiple physical variables per node, groups can be formed by grouping the equations per node.
- In structured matrices, such as those from PDEs on regular grids, a partitioning can be based on the physical domain.
- On parallel environments, it is natural to let the partitioning coincide with the division of variables over the processors.

Jacobi preconditioners require very little storage and they are easy both to construct and to apply. Additionally they can be have straightforward parallel implementation. On the other hand more sophisticated preconditioners usually yield a larger improvement.

1.3.2 Incomplete Factorization preconditioners

A broad class of preconditioners is based on incomplete factorizations of the coefficient matrix. A factorization is called incomplete if during the factorization process certain elements are ignored. These elements are called *fill* elements and are nonzero elements that arise during the factorization in positions where the original matrix had a zero. Discarding fill elements makes the building phase of the preconditioner almost cheap when compared to complete factorizations $A = LU$; moreover, complete factorizations have one major problem when applied to sparse matrices: even if the system matrix A is sparse, L and U are not sparse in general and so could not be stored due to the excessive memory requirements. However incomplete factorizations may break down (division by zero pivot. In this cases some workarounds have been proposed) or result in undefined matrices.

An important consideration for incomplete factorization preconditioners is the cost of the factorization process itself. Even if the incomplete factorization exists, the number of operations involved in creating it is at least as much as for solving a system with such a coefficient matrix, so the cost may be the same as for one or more iterations of the method.

In some cases it is possible to take advantage of the fact that the same preconditioner can be used to solve more than one system, and thus the construction cost has a lower impact.

Incomplete factorizations can be given in various forms. If $M = LU$ (with L and U nonsingular triangular matrices), solving a system proceeds in the usual way, but often incomplete factorizations are given as $M = (D + L)D^{-1}(D + U)$ (with D diagonal and L and U now strictly triangular matrices). In that case, either of the following equivalent formulations could be used for $Mx = y$:

$$(D + L)z = y, \qquad (I + D^{-1}U)x = z$$

or

$$(I + LD^{-1})z = y, \qquad (D + U)x = z.$$

In either case, the diagonal elements are used twice and, since only divisions with D are performed, it is practical to store D^{-1} instead of D.

The most common type of incomplete factorization is based on taking a set S of matrix positions, and keep all the positions outside this set equal to zero during the factorization. The resulting factorization is incomplete in the sense that fill is suppressed.

The set S is usually chosen to encompass all the positions (i, j) for which $a_{i,j} \neq zero$. A position that is zero in A but not so in a complete factorization

is called a fill position and, if it is outside S, the fill there is said to be discarded. When S is chosen to coincide with the set of nonzero positions of A the factorization is called Incomplete Factorization of level 0 (ILU(0)).

An incomplete factorization can be formally described as

$$\text{for each } k, i, j > k: \quad a_{i,j} \leftarrow \begin{cases} a_{i,j} - a_{i,k} a_{k,k}^{-1} a_{k,j} & \text{if } (i,j) \in S \\ a_{i,j} & \text{otherwise} \end{cases}$$

For the ILU(0) method, the incomplete factorization produces no nonzero elements beyond the sparsity structure of the original matrix so that the preconditioner at most takes exactly as much space to store as the original matrix. In a simplified version of ILU(0), called ILU(D) not only fill elements are prohibited, but only the diagonal elements are altered.

Splitting the coefficient matrix into its diagonal, lower triangular and upper triangular parts as $A = D_A + L_A + U_A$, the preconditioner can be written as $M = (D + L_A)D^{-1}(D + U_A)$ where D is the diagonal matrix containing the pivots generated. Since the upper and lower triangle of the system matrix remain unchanged in the preconditioner matrix, only storage space for D is needed. As usual storing D^{-1} instead of D if the most efficient choice.

Chapter 2

The PSBLAS Library

Contents

This section presents the software architecture and some implementation features of a library of Basic Linear Algebra Subroutines for parallel sparse applications, namely Parallel Sparse BLAS (PSBLAS) [39],which is a project that has been prompted by the appearance of the proposal for serial sparse BLAS [30].

The PSBLAS library, developed with the aim to facilitate the parallelization of computationally intensive scientific applications, is designed to address parallel implementation of iterative solvers for sparse linear systems through the distributed memory paradigm. It includes routines for multiplying sparse matrices by dense matrices, solving block diagonal systems with triangular

diagonal entries, preprocessing sparse matrices, and contains additional routines for dense matrix operations. The current implementation of PSBLAS addresses a distributed memory execution model operating with message passing. However, the overall design does not preclude different implementation paradigms, such as those based on a shared memory model.

The PSBLAS library is internally implemented in a mixture of Fortran 77 and Fortran 95 [54] programming languages. A similar approach has been advocated by a number of authors, e.g. [52]. Moreover, the Fortran 95 facilities for dynamic memory management and interface overloading greatly enhance the usability of the PSBLAS subroutines. In this way, the library can take care of runtime memory requirements that are quite difficult or even impossible to predict at implementation or compilation time. The following presentation of the PSBLAS library follows the general structure of the proposal for serial Sparse BLAS [30], which in its turn is based on the proposal for BLAS on dense matrices [26, 27, 50].

The applicability of sparse iterative solvers to many different areas causes some terminology problems because the same concept may be denoted through different names depending on the application area. The PSBLAS features presented in this section will be discussed mainly in terms of finite difference discretizations of Partial Differential Equations (PDEs). However, the scope of the library is wider than that: for example, it can be applied to finite element discretizations of PDEs, and even to different classes of problems such as nonlinear optimization, for example in optimal control problems.

The design of a solver for sparse linear systems is driven by many conflicting objectives, such as limiting occupation of storage resources, exploiting regularities in the input data, exploiting hardware characteristics of the parallel platform. To achieve an optimal communication to computation ratio on distributed memory machines it is essential to keep the *data locality* as high as possible; this can be done through an appropriate data allocation strategy. The choice of the preconditioner is another very important factor that affects efficiency of the implemented application. Optimal data distribution requirements for a given preconditioner may conflict with distribution requirements of the rest of the solver. Finding the optimal trade-off may be very difficult because it is application dependent. Possible solution to these problems and other important inputs to the development of the PSBLAS software package has come from an established experience in applying the PSBLAS solvers to computational fluid dynamics applications (see chapter 6.2). As a typical case, a complete simulation of an internal combustion engine can easily scale up to some millions of variables and, thus, represents a challenging testbed to verify the effectiveness of the PSBLAS library implementation.

2.1 General overview

The PSBLAS library is designed to handle the implementation of iterative solvers for sparse linear systems on distributed memory parallel computers. The system coefficient matrix A must be square; it may be real or complex, nonsymmetric, and its sparsity pattern needs not to be symmetric. The serial computation parts are based on the serial sparse BLAS, so that any extension made to the data structures of the serial kernels is available to the parallel version. The overall design and parallelization strategy have been influenced by the structure of the ScaLAPACK parallel library [21]. The layered structure of the PSBLAS library is shown in figure 2.1 ; lower layers of the library indicate an encapsulation relationship with upper layers. The ongoing discussion focuses on the Fortran 95 layer immediately below the application layer; two examples of iterative solvers built through the PSBLAS routines, will be also given in Section 2.6. The serial parts of the computation on each process are executed through calls to the serial sparse BLAS subroutines. In a similar way, the inter-process message exchanges are implemented through the Basic Linear Algebra Communication Subroutines (BLACS) library [24] that guarantees a portable and efficient communication layer. The Message Passing Interface code is encapsulated within the BLACS layer. However, in some cases, MPI routines are directly used either to improve efficiency or to implement communication patterns for which the BLACS package doesn't provide any method.

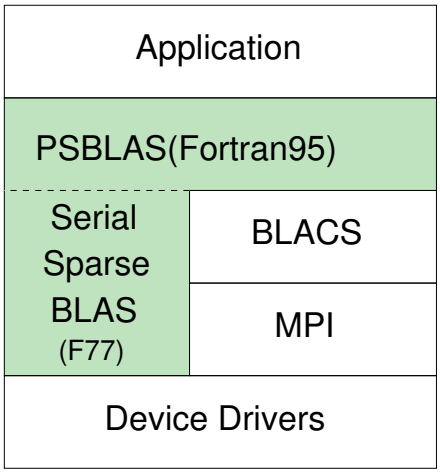

Figure 2.1: PSBLAS library components hierarchy.

The PSBLAS library consists of two classes of subroutines that is, the *computational routines* and the *auxiliary routines*. The computational routine

set includes:

- Sparse matrix by dense matrix product;
- Sparse triangular systems solution for block diagonal matrices;
- Vector and matrix norms;
- Dense matrix sums;
- Dot products.

The auxiliary routine set includes:

- Communication descriptors allocation;
- Dense and sparse matrix allocation;
- Dense and sparse matrix build and update;
- Sparse matrix and data distribution preprocessing.

The following naming scheme has been adopted for all the symbols internally defined in the PSBLAS software package:

- all the symbols (i.e. subroutine names, data types...) are prefixed by `psb_`
- all the data type names are suffixed by `_type`
- all the constant values are suffixed by `_`
- all the subroutine names follow the rule `psb_xxname` where `xx` can be either:
  - `ge`: the routine is related to dense data,
  - `sp`: the routine is related to sparse data,
  - `cd`: the routine is related to communication descriptor (see 2.2.2).

  For example the `psb_geins`, `psb_spins` and `psb_cdins` perform the same action (see 2.5) on dense matrices, sparse matrices and communication descriptors respectively. Interface overloading allows the usage of the same subroutine interfaces for both real and complex data.

2.2 Main PSBLAS data structures

2.2.1 Library design choices

In any distributed memory application, the data structures that represent the partition of the computational problem are essential to the viability and efficiency of the entire parallel implementation. The criteria guiding the decomposition choices are:

1. maximizing load balancing,
2. minimizing communication costs,
3. optimizing the efficiency of the serial computation parts.

For *dense* linear algebra algorithms the ScaLAPACK library [21] has demonstrated that a block-cyclic distribution of the row and column index spaces is general and powerful enough to achieve a good compromise among the various factors that affect parallel computation performance. PSBLAS library addresses parallel *sparse* iterative solvers typically arising in the numerical solution of PDEs. In these instances, it is necessary to pay special attention to the structure of the problem from which the application originates. The nonzero pattern of a matrix arising from the discretization of a PDE is influenced by various factors, such as the shape of the domain, the discretization strategy, and the equation/unknown ordering. The matrix itself can be interpreted as the adjacency matrix of the graph associated with the discretization mesh; this characteristic will be discussed in section 2.2.5. The allocation of the coefficient matrix for the linear system is based on the "owner computes" rule: the associated variable of each mesh point is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations. This allocation strategy is equivalent to a partition of the discretization mesh into *sub-domains*. PSBLAS routines support any distribution that keeps together the coefficients of each matrix row; there are no other constraints on the variable assignment. The available distribution strategies include data distributions commonly used in ScaLAPACK such as `CYCLIC(N)` and `BLOCK`, as well as completely arbitrary assignments of equation indices to processes. Dense vectors conform to sparse matrices, that is, the entries of a vector follow the same distribution of the matrix rows. It is never required that the entire system matrix is available on a single node and efficiency is obviously improved in the case where each node generates its own portion of the system. However, it is possible to hold the entire matrix in one process and distribute it explicitly, even though the resulting bottleneck would make this option unattractive in most cases. The storage

scheme used for the computational parts pertaining to each process conforms to the storage formats defined in the serial sparse BLAS proposal [30]. The data structures that describe the local matrix storage are kept separated from those used for representing the communication pattern. This choice satisfies the encapsulation relations among the PSBLAS layers as described previously.

2.2.2 Communication descriptors

Once the distributed sparse matrix has been allocated and built, it is necessary to arrange the data structures that will be used to perform inter-process communications during the execution of routines such as matrix-vector products, preconditioners and scalar products. The detailed contents of these data structures depend on the sparsity pattern of the coefficient matrix. This index space is classified according to the user defined assignment of its elements to the processes of the parallel machine. The PSBLAS computational model implies that the data allocation on the parallel distributed memory machine is guided by the structure of the physical model, and specifically by the discretization mesh of the PDE. Each point of the discretization mesh will have (at least) one associated equation/variable, and therefore one index. Point i *depends* on point j if the equation for a variable associated with i contains a term in j, or equivalently if $a_{ij} \neq 0$. After the partition of the discretization mesh into *sub-domains* assigned to the parallel processes, the points of a given sub-domain can be classified as following:

Internal. An internal point of a given subdomain *depends* only on points of the same subdomain. If all points of a domain are assigned to one process, then a computational step (e.g., a matrix-vector product) of the equations associated with the internal points requires no data items from other subdomains and no communications.

Boundary. A point of a given subdomain is a boundary point if it *depends* on points belonging to other subdomains.

Halo. A halo point for a given subdomain is a point belonging to another subdomain such that there is a boundary point which *depends* on it. Whenever performing a computational step, such as a matrix-vector product, the values associated with halo points are requested from other subdomains. A boundary point of a given subdomain is a halo point for (at least) another subdomain; therefore the cardinality of the boundary points set denotes the amount of data sent to other subdomains.

Overlap. An overlap point is a boundary point assigned to multiple subdomains. Any operation that involves an overlap point has to be replicated for each assignment. This may be acceptable, for example to accelerate the convergence rate of the overall iterative method through an improvement of the preconditioning task [37].

The sets of internal, boundary and halo points for a given subdomain are denoted by $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{H}$ respectively. Each subdomain is assigned to one process; each process usually owns one subdomain in which case the number of rows in the local sparse matrix is $|\mathcal{I}_i| + |\mathcal{B}_i|$, and the number of local columns (i.e. those for which there exists at least one non-zero entry in the local rows) is $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$. The representation of the points of a subdomain assigned to a process is stored into integer arrays; the internal format is described in more detail in [19]. Upon each invocation of a PSBLAS subroutine, it is assumed that only one set of communication descriptors is active. This assumption is not too restrictive in view of commonly addressed applications. Indeed, if the library is used in the context of a PDE solver, often have multiple equations are defined on the same discretization mesh, so that the coefficient matrices of the linear systems have the same sparsity pattern. The definition of the communication descriptors is the following.

MATRIX_DATA A vector containing some general information, such as the descriptor status, the size of the global matrix, the number of local rows and columns and the BLACS communication context.

HALO_INDEX The local list of the indices of halo points that have to be exchanged with other processes. For each process, this list contains the process identifier, the number and indices of points to be received, the number and indices of points to be sent.

OVERLAP_INDEX A (local) list of the overlap points, organized in groups like the halo index descriptor; the format is identical to that of the halo descriptor.

OVERLAP_ELEM A (local) list containing for each overlap point its local index and the number of processes sharing it. This information is implicitly available in `OVERLAP_INDEX`; it is computed and stored separately at initialization time for efficiency reasons.

Two auxiliary arrays are used to keep track of the mapping between local and global indices. The Fortran 95 language allows a convenient packing of the necessary data structures inside a single *derived data type* as reported in figure 2.2.

```
type psb_desc_type
   integer, pointer :: matrix_data(:), halo_index(:)
   integer, pointer :: overlap_elem(:), overlap_index(:)
   integer, pointer :: loc_to_glob(:), glob_to_loc(:)
end type psb_desc_type
```

Figure 2.2: The PSBLAS defined data type that contains the communication descriptor.

2.2.3 Sparse matrix storage

Assembling a sparse matrix requires that the user defines matrix entries in terms of the global equation numbering. Then, each process in the parallel machine builds the sparse matrix rows that are assigned to it by the user by means of the `psb_spins` subroutine described below. Once the build step is completed, the local part of the matrix undergoes a preprocessing operation. During this step, performed through the `psb_spasb` subroutine, the global numbering scheme is converted into the local numbering scheme, and the local sparse matrix representation is converted to a format suitable for subsequent computations.

The serial sparse BLAS routines [18, 30] are used to carry out the internal storage conversion step and all other operations involving local sparse matrix computations. The paper [30] contains a detailed discussion about the rationale for the sparse matrix representation shown below. In particular, it describes the format of the permutation vectors `pr` and `pl`. These permutations arise in various sparse storage format conversions that may impose a renumbering of the local equations and variables to achieve the desired runtime efficiency. In PSBLAS, the sparse matrix storage conforms to the Fortran 95 implementation of the serial sparse data structures reported in figure 2.3.

Complex matrices have a similar structure, with the appropriate type declaration for member `aspk`. At the moment the PSBLAS library provides the possibility to store sparse matrices in the Compressed Sparse Row (CSR), Coordinate (COO) and Jagged Diagonal (JAD) formats (see section 3.1.2, 3.1.1 and 3.1.3) while the Blocked CSR, described in chapter 3, is being integrated. However the `psb_spmat_type` data structure has been designed to be flexible enough to easily allow the implementation of other storage formats. In this sense, the content of the `ia1`, `ia2` and `aspk` arrays is interpreted accordingly to the content of the `fida` and `infoa` records. The implementation of the

```
type psb_dspmat_type
   integer      :: m, k
   character    :: fida(5)
   character    :: descra(10)
   integer      :: infoa(10)
   real(kind(1.d0)), pointer :: aspk(:)
   integer, pointer :: ia1(:), ia2(:), pr(:), pl(:)
end type psb_dspmat_type
```

Figure 2.3: The PSBLAS defined data type that contains a sparse matrix.

BCSR storage format described in chapter 3 is an example of it is possible to use the `psb_spmat_type` to integrate new matrix representations into the PSBLAS library.

2.2.4 Preconditioner data structure

PSBLAS-2.0 offers the possibility to use many different types of preconditioning schemes. Besides the simple well known preconditioners like Diagonal Scaling or Block Jacobi (with ILU(0) incomplete factorization) also more complex preconditioning methods are implemented like the Additive Schwarz and Two-Level ones (see respectively chapters 4 and 5). A preconditioner is held in the `psb_prec_type` data structure which depends on the `psb_base_prec` reported in figure 2.4. The `psb_base_prec` data type may contain a simple preconditioning matrix with the associated communication descriptor which may be different than the system communication descriptor in the case of parallel preconditioners like the Additive Schwarz one. Then the `psb_prec_type` may contain more than one preconditioning matrix like in the case of Two-Level (in general Multi-Level) preconditioners. The user can choose the type of preconditioner to be used by means of the `psb_precset` subroutine; once the type of preconditioning method is specified, along with all the parameters that characterize it, the preconditioner data structure can be built using the `psb_precbuild` subroutine. This data structure wants to be flexible enough to easily allow the implementation of new kind of preconditioners. The values contained in the `iprcparm` and `dprcparm` define tha type of preconditioner along with all the parameters related to it; thus, `iprcparm` and `dprcparm` define how the other records have to be interpreted. Chapters 4 and 5 show that this data structure is general enough to implement Additive Schwarz

and Two-Level preconditioners.

```
 type psb_base_prec

   type(psb_spmat_type), pointer  :: av(:) => null()
   real(kind(1.d0)), pointer      :: d(:)  => null()
   type(psb_desc_type), pointer   :: desc_data => null()
   integer, pointer               :: iprcparm(:) => null()
   real(kind(1.d0)), pointer      :: dprcparm(:) => null()
   integer, pointer               :: perm(:)  => null()
   integer, pointer               :: mlia(:)  => null()
   integer, pointer               :: invperm(:) => null()
   integer, pointer               :: nlaggr(:)  => null()
   type(psb_spmat_type), pointer  :: aorig    => null()
   real(kind(1.d0)), pointer      :: dorig(:) => null()

end type psb_base_prec

 type psb_prec_type
   type(psb_base_prec), pointer  :: baseprecv(:) => null()
   integer                       :: prec, base_prec
end type psb_prec_type
```

Figure 2.4: The PSBLAS defined data type that contains a preconditioner.

2.2.5 Data allocation strategies and graph partitioning

It has been previously mentioned that the user can take any decision about the allocation of matrix rows and associated variables to processes; indeed, some experimental tests were based on random assignments. However each allocation choice will result in a different runtime efficiency. It can be assumed that a good runtime efficiency, in PSBLAS, can be achieved through the choice of a data allocation that aims at minimizing the execution time of matrix-vector products. To motivate this assessment, it is necessary to consider the computational and communication costs of target applications of the PSBLAS library.

Modern iterative solvers are typically based on Krylov subspace approximations (see section 1.2). In most instances, the cost of each iteration is constant across iterations. It is made up of matrix-vector products, preconditioning

operations, scalar products and dense vector sums. By analyzing these operations it can be stated that:

- Dense vector sums do not involve any communication.
- Scalar products have a communication cost that is determined by the number of processes in the parallel machine.
- Many preconditioners, such as Jacobi, SOR, diagonal scaling and basic ILU variants, either do not require inter-process communications or they are associated with matrices having the same sparsity pattern as that of the coefficient matrix A.

The above observations imply that the only balance criterion for dense vector operations is that the vector entries should be evenly distributed among processes.

Incomplete factorization preconditioners that operate locally are also implemented inside PSBLAS, because they usually give a good compromise between parallelization efficiency and preconditioning effectiveness in terms of reduction of the number of iterations. It would be also possible to implement a global factorization preconditioner by using our communication descriptors, provided that some constraints are imposed on the fill-in generation. However, this choice would require heavy modifications on the triangular matrix T storage format. Moreover, the performance of the triangular system solution would be much lower. In summary, the applicability and efficacy of a global factorization preconditioner do not seem general enough to warrant its implementation.

As a consequence of previous observations, with little loss of generality, it makes sense to direct optimization efforts towards the improvement of the parallel implementation of the matrix-vector product (see also section 3.2). The search for the optimal data distribution for this basic operation can be modeled as a weighted graph partitioning problem because the coefficient matrix can be interpreted as the adjacency matrix of the graph associated with the discretization mesh.

Each node of the graph models one row in the coefficient matrix; its weight equals the number of nonzero coefficients, as this is proportional to the computational cost of evaluating its contribution to the matrix-vector product. Each arc of the graph models a nonzero coefficient that needs to acquire the value of one variable to complete the computation associated with an equation. It can be assumed that the cost of communicating a variable value is constant, so that all arcs have the same (unitary) weight.

The optimality criteria for the graph partitioning problem are:

1. The total weight of the graph nodes should be evenly distributed.

2. The weight of the edges crossing partition boundaries should be minimized.

If the variance degree is not too large, or equivalently if the graph structure is sufficiently regular, the node weight criterion induces a distribution of the graph nodes that balances performance of vector operations. Each vector operation adds a fixed number of floating point operations per matrix row, and this is equivalent to adding a constant value to each node weight.

The general graph partition problem is $\mathcal{NP}$-complete [43] and, thus, a vast body of heuristic has been developed to obtain reasonably good partitions in an effective way (see [1, 45, 61], and references therein). PSBLAS can be interfaced with popular tools such as Metis [1] with obvious benefits in performance. However, the use of a graph partition tool is not strictly required by PSBLAS. In fact, for most problems of interest to the PSBLAS library, a surface to volume effect takes place, so that even a `BLOCK` distribution may be satisfactory. Moreover, it is likely that the partition of the discretization mesh can be guided by physical considerations. Note that many graph partitioning algorithms work on undirected graphs, which would correspond to matrices with a symmetric pattern, but this is not required by PSBLAS. In these instances, it would be appropriate (even if simplistic) to apply the partition tool to the graph associated with the nonzero pattern of $A + A^T$, and to apply the resulting distribution to the original matrix.

For these reasons, the PSBLAS library is designed with the intention of letting the user control the entire specification of the data allocation. The only constraint is that the data allocation strategy satisfies the "owner computes" paradigm discussed in section 2.2.1.

2.3 PSBLAS operations

For the description of the operations in the PSBLAS library the following definitions and assumptions are used.

A is a sparse matrix distributed by rows as described above;

T is a sparse matrix with triangular blocks on the main diagonal and zeros elsewhere; it is distributed by rows conforming to A;

B, C, X, Y are dense matrices distributed by rows conforming to A;

x, y are dense vectors distributed by rows conforming to A;

α, β are scalars;

P, P_R, P_C are block diagonal permutation matrices;

D is a diagonal matrix distributed by rows conforming to A.

The block diagonal form of T is due to the choice of a factorization preconditioner that works only on the part of the matrix local to each process. Hence, each block extends across the rows of the matrix assigned to a process. This preconditioner is usually called *block Jacobi* (sometimes *block ILU*) or *overlapping additive Schwartz*(see chapter 4) depending on the existence of overlap among sub-domains.

The permutation matrices P, P_R, P_C are in block diagonal form too, because they depend on the storage format chosen for the part of A local to each process. Indeed, they do not appear explicitly in calling sequences because they are encapsulated into the local part of A, as discussed in section 2.2.3.

The main operations in the PSBLAS library are:

- Matrix-matrix products

$$\begin{aligned} C &\leftarrow \alpha P_R A P_C B + \beta C \\ C &\leftarrow \alpha P_R A^T P_C B + \beta C \end{aligned}$$

- Triangular system solutions

$$\begin{aligned} C &\leftarrow \alpha P_R T^{-1} P_C B + \beta C \\ C &\leftarrow \alpha D P_R T^{-1} P_C B + \beta C \\ C &\leftarrow \alpha P_R T^{-1} P_C D B + \beta C \\ C &\leftarrow \alpha P_R T^{-T} P_C B + \beta C \\ C &\leftarrow \alpha D P_R T^{-T} P_C B + \beta C \\ C &\leftarrow \alpha P_R T^{-T} P_C D B + \beta C \end{aligned}$$

- Matrix sums

$$Y \leftarrow \alpha X + \beta Y$$

- Scalar products

$$x^T y \qquad \text{or} \quad x^H y$$

- Dense vector 1, 2 and infinity norms

$$\|x\|$$

- Sparse matrix infinity norm

$$\|A\|_\infty$$

- Halo data communications
- Overlap data updates

The default behavior of the subroutines is to keep all vector entries consistent across all subroutine calls, although this may be overridden for efficiency reasons. As an example, the application of an ILU preconditioner requires two successive calls to the triangular system solution routine: the former for the L part, and the latter for the U part of the preconditioner. As the preconditioning step is a local operation, it can be chosen to restore consistency on overlap points only after the second call to the triangular system solver routine.

2.4 Computational subroutines

Computational subroutines have been developed based on the assumption that they must be by all processes. Even if there is there are no explicit barrier constructs all computational subroutines employ global communications for error checking, thus providing synchronization. In all the computational subroutine interfaces, `desc_a` is the communication descriptor of type `psb_desc_type` as described in section 2.2.2 while the `info` argument (present in all of the PSBLAS subroutine interfaces) is used for error handling as described in section 2.8.

The input parameters are ordered in such a way that the parameters occurring after `info` are optional. Sparse matrices `A` and `T` must be declared of type `psb_spmat_type`. Input scalars such as `ALPHA` and `BETA` must have the same value on all processes; dense matrices and vectors such as `X` and `x` must have the `POINTER` attribute.

psb_gedot Computes the dot product (x, y) of two vectors

```
dot = psb_gedot(x, y, desc_a, info)
```

psb_geaxpby Computes

$$Y \leftarrow \alpha X + \beta Y$$

```
call psb_geaxpby(alpha, x, beta, y, desc_a, info)
```

psb_geamax Computes the maximum absolute value, or infinity norm of a vector

$$amax \leftarrow \max_i \{|X(i)|\} = \|X\|_\infty.$$

```
amax =  psb_geamax(x, desc_a, info)
```

psb_geasum Computes the 1-norm of a vector

$$asum \leftarrow \sum_i |X(i)| = \|X\|_1.$$

```
asum = psb_geasum(x, desc_a, info)
```

psb_genrm2 Computes the 2-norm of a vector

$$nrm2 \leftarrow \|X\|_2.$$

```
nrm2 = psb_genrm2(x, desc_a, info)
```

psb_spnrmi Computes the infinity-norm of a distributed sparse matrix

$$nrmi \leftarrow \|A\|_\infty.$$

```
nrmi = psb_spnrmi(a, desc_a, info)
```

psb_spmm Computes the sparse matrix by dense matrix product

$$\begin{array}{rcl} Y & \leftarrow & \alpha P_R A P_C X + \beta Y \\ Y & \leftarrow & \alpha P_R A^T P_C X + \beta Y \end{array}$$

The permutation matrices P_R and P_C are local, as they handle all details of the local storage format. They do not appear explicitly in the following call format because they are encapsulated into A, according to the derived data type discussed in Section 2.2.3.

```
call psb_spmm(alpha, a, x, beta, y, desc_a, info, trans)
```

psb_spsm Computes the triangular system solution:

$$\begin{array}{rcl} Y & \leftarrow & \alpha P_R T^{-1} P_C X + \beta Y \\ Y & \leftarrow & \alpha D P_R T^{-1} P_C X + \beta Y \\ Y & \leftarrow & \alpha P_R T^{-1} P_C D X + \beta Y \\ Y & \leftarrow & \alpha P_R T^{-T} P_C X + \beta Y \\ Y & \leftarrow & \alpha D P_R T^{-T} P_C X + \beta Y \\ Y & \leftarrow & \alpha P_R T^{-T} P_C D X + \beta Y \end{array}$$

The triangular sparse matrix T is block diagonal. This is equivalent to the application of local ILU or IC preconditioning.

```
call psb_spsm(alpha, t, x, beta, y, desc_a, info,
              trans, unitd, iopt, d)
```

The character `unitd` parameter denotes how to use the diagonal D (stored as a vector), which can be assumed unitary (and thus ignored), or applied on the left or on the right. The *iopt* parameter denotes whether the consistency of the overlap points for vector X has to be enforced or not.

2.5 Auxiliary subroutines

The PSBLAS library contains a set of tools that define the parallel data structure, and allocate and assemble the matrices involved in the computation. The goal is to encapsulate the low-level details of the internal storage of the communication descriptors and sparse matrices. The requirements placed on the user that wants to write applications in PSBLAS are the following.

1. The allocation of the index space among the processes is defined by means of the subroutine `parts` described below. Alternatively, a vector v of size n, where n is the matrix size, can be used such that $v(i)$ contains the identifier of the process that owns point i.

2. The coefficients of the sparse matrix are given in terms of global numbering.

The choice of a subroutine argument aims to guarantee maximal flexibility; the user can define an arbitrary data allocation scheme and experiment various partition strategies in a simple way. The PSBLAS package includes some predefined subroutines for simple distributions such as `BLOCK` and `CYCLIC`. The coefficients of the linear system should be generated in a very simple format (currently, Coordinate) that is handled through the subroutine `psb_spins`. The library will manage other storage schemes through the preprocessing facilities of the serial sparse BLAS [30].

For any application, the first PSBLAS routine invoked must be `psb_cdall` that allocates a communication descriptor data structure and initializes the parallel environment. At the time this subroutine is called, it is assumed that the application has already initialized the BLACS communication environment; each PSBLAS process will identify itself by means of its BLACS task index.

Auxiliary subroutines must be called by all processes, with the exception of the insertion routines `psb_cdins`, `psb_spins` and `psb_geins` (equivalent to the `psb_spins` for dense matrices). These routines are called independently by each process to act on the local parts of the relevant sparse or dense

matrices. The processes will synchronize upon the subsequent (and required) call to psb_cdasb, psb_spasb and psb_geasb routines.

The auxiliary subroutines currently include:

parts This is a user provided subroutine. The index space allocation among the processes is obtained through the library call

```
call parts(g_idx,procs,nprocs)
```

where g_idx is the input global index from the library routine, procs(*) is the output vector containing the indices of the BLACS task(s) owing the given index, and nprocs is the number of valid entries in procs. If nprocs > 1, there is an overlap point.

psb_cdall Allocates and initializes communication descriptors data structures on the basis of the user information that is provided through the subroutine PARTS;

```
call psb_cdall(m, n, parts, icontxt, desc_a, info)
call psb_cdall(m, n, v, icontxt, desc_a, info)
```

m and n are the global matrix dimensions: they are kept separate, even though currently only square matrices are supported. The parameter icontxt is the BLACS communication context returned by the BLACS environment initialization routine. This routine can be fed with an integer vector v specifying the process that owns each variable instead of the parts subroutine.

psb_spall Allocates data structures for Global Sparse Matrix. The user may also provide an estimate of the number of nonzero elements that have to be allocated for the local part of the sparse matrix.

```
call psb_spall(a, desc_a, info, nnz)
```

psb_geall Allocates a dense matrix; dense matrices must be declared with the pointer attribute.

```
call psb_geall(m, n, x, desc_a, info)
```

psb_cdins Updates the communication descriptor accordingly to the communication patterns defined by a set of points. The ia and ja arrays contain respectively the row and column indices of the nnz points to be inserted.

```
call psb_cdins(nnz, ia, ja, desc_a, info, is, js)
```

psb_spins Inserts a sparse block into a sparse matrix. A local sparse matrix in coordinate (COO) format is inserted into the local part of the matrix A. This sparse is passed through the `ia`,`ja` and `val` arguments containing respectively the row indices, the column indices and the values of the block entries. It is not necessary to pass a complete matrix block to this routine. For example, the user that wants to implement a finite element application can denote the equations element by element without any ordering constraint. PSBLAS is able to keep track of contributions from different elements into the same equation (row of the sparse matrix). Multiple contributions to the same matrix entry can be treated by summing their values, by keeping just one of the specified values and ignoring the others, or by raising an error condition, under the user control.[1]

```
call psb_spins(nz, ia, ja, val, a, desc_a, info, is, js)
```

psb_geins Inserts a dense block into a dense matrix.

```
call psb_dins(m, n, x, ix, jx, blck, desc_a, info, is, js)
```

psb_cdasb Assembles the communication descriptor. It processes the communication descriptor to put it into the final format that is suitable for the computational routines.

```
call psb_cdasb(desc_a, info)
```

psb_spasb Assembles Sparse Matrix. It processes the sparse matrix to put it into the final format that is suitable for the computational routines. This routine checks for errors that may have occurred during the insertion phase.

```
call psb_spasb(a, desc_a, info)
```

psb_geasb Analogous to `psb_spasb` for a dense matrix.

```
call psb_geasb(x, desc_a, info)
```

[1] This is done through the serial preprocessing routines; since this behavior is not strictly specified in the serial sparse BLAS, it can be viewed as an extension to that specification.

psb_cdfree Deallocates the memory associated with the components of a descriptor data structure.

```
call psb_cdfree(desc_a, info)
```

psb_spfree Deallocates the memory associated with the components of a sparse matrix.

```
call psb_spfree(a, desc_a, info)
```

psb_gefree Deallocates a dense matrix.

```
call psb_gefree(x, desc_a, info)
```

psb_spreinit Reinits a sparse matrix. This operation allows the regeneration of a matrix on the base of information stored in a previous matrix building.

```
call psb_spreinit(a, desc_a, info)
```

Substantial savings in the time needed to build a sparse matrix may be achieved by considering that for many applications the same problem has to be solved repeatedly, for example in successive time steps. If the topology of the discretization mesh doesn't change from on time step to another, the system matrices will have the same sparsity pattern. In this case extra information may be stored during the insert and assembly steps of the first problem solution to be reused in successive system solution allowing to perform the same steps in a shorter time.

During their existence, a sparse matrix and a communication descriptor may be in different states. A sparse matrix may be in the `bld`, `asb` or `upd` states while a descriptor may be in the `bld` or `asb` ones. Provided that all the computational subroutines may be invoked only when both of them are in the `asb` state, there are different paths to reach these states by means of the insert and assembly routines to provide the user the higher flexibility as possible. Namely there are to paths: one where the matrix and descriptor constructions are handled separately and another one to build both of them at once.

Figure 2.5 reports the typical layout of a PSBLAS application and shows how the sparse matrix and descriptor data structures can be built and which state transitions are determined by the invocation of some auxiliary subroutines (note that either 2.1 or 2.2 step is performed):

1 Initialize the communication descriptors with `psb_cdall`; initialize the sparse matrix with `psb_spall` and right hand side with `psb_geall`. After this step, both the sparse matrix and the communication descriptor are in the `bld` state.

2.1 This step is associated to the right branch on figure 2.5:

a Loop on all mesh points owned by the current process, build their equations and insert them into the communication descriptor through calls to the `psb_cdins` subroutine;

b Assemble the communication descriptor through a call to the `psb_cdasb` subroutine. At the end of this step the descriptor is in the `asb` state;

c Loop on all mesh points owned by the current process (not necessarily in the same order as in step a), build their equations and insert them into the sparse matrix and right hand side through calls to the `psb_spins` and `psb_geins` subroutines,

d Assemble the sparse matrix and the right hand side through calls to the `psb_spasb` and `psb_geasb` subroutine. At the end of this step the sparse matrix is in the `asb` state;

2.2 This step is associated to the left branch on figure 2.5:

a Loop on all mesh points owned by the current process, build their equations and insert them into the communication descriptor, sparse matrix and right hand side through calls to `psb_spins` and `psb_geins`. If the descriptor is in the `bld` state instead of the `asb` one (which is the case of the left branch in figure 2.5), the `psb_spins` also takes care of inserting the points into it;

b Assemble the communication descriptor, sparse matrix and right hand side through calls to the `psb_cdasb` , the `psb_spasb` and the `psb_geasb`. At the end of this step both the descriptor and the sparse matrix are in the `asb` state;

3 Compute the preconditioner and call the iterative method.

4 If a system with the same sparsity pattern must be solved, the system matrix may be reinitialized through a call to the `psb_spreinit` subroutine. At the end of this step the matrix is in the `upd` state

5 Loop on all the mesh points owned by the current process in the same order as in step 2.1.c or 2.2.a, build their equations and insert them into

the sparse matrix and right hand side through calls to the psb_spins and psb_geins subroutines,

6 Assemble the sparse matrix and the right hand side through calls to the psb_spasb and psb_geasb subroutine. At the end of this step the sparse matrix is in the asb state; go to step 3.

The order in which the mesh points are visited in step 2.1.c or 2.2.a is arbitrary. This is useful for finite element applications, where it may be desirable to loop over the elements, rather than over the equations. As an example, consider the sample application shown in Section 5 of [52]: the Poisson matrix assemble loop in function assemble_matrix_a runs through the elements. Recasting the sample code into PSBLAS calls, yields:

```
DO iel=1,ael%nel
    blck = give_matrix_e(ael%el(iel),which)
    ! assume IMIN is the lowest row index among ael%index(i,iel)
    CALL psb_spins(blck%info(psb_nnz_),
                   blck%ia1,blck%ia2,a,desc_a,info)
ENDDO
CALL psb_spasb(a,desc_a,info)
```

This example assumes that the original function give_matrix_e has been rewritten to return a sparse matrix in coordinate format instead of a dense matrix as in [52].

If the data distribution is independent of the discretization mesh structure, for example when using a BLOCK distribution, the above application structure has no serial bottlenecks. If a graph partitioning package is used, the cost of the setup phase depends on the graph partitioning routine. If the subroutine is serial, there is a serial bottleneck both in terms of processing time and memory space, because one of the processes will hold the structure of the entire discretization mesh. For many applications this would not be a serious drawback, because the linear solver itself is a single step in an outer solution algorithm, and often many (if not all) consecutive steps share the same mesh topology.

The serial graph partitioning approach is not very suitable to applications that use adaptive meshes with fast rates of change; extensions to PSBLAS for such applications are currently being investigated [40].

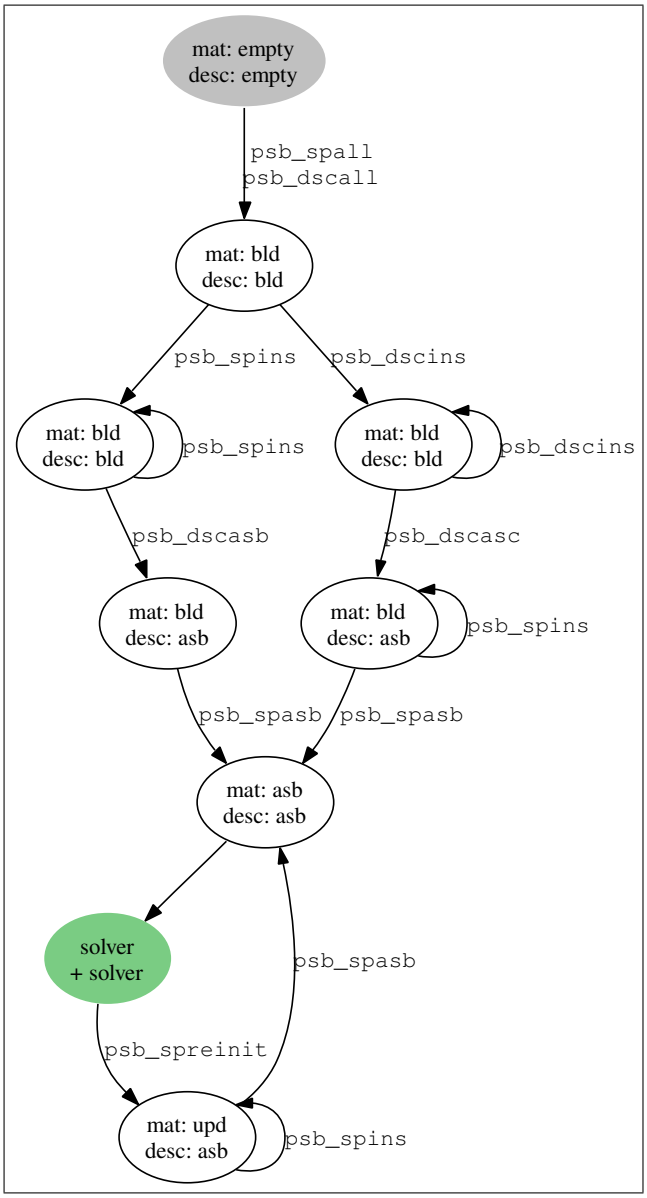

Figure 2.5: PSBLAS data structures building steps.

2.6 Iterative methods

To illustrate the use of the library routines, the templates for the CG and Bi-CGSTAB methods from [7], with local ILU preconditioning and normwise backward error stopping criterion [3] are reported in tables 2.1 and 2.2. The examples show the high readability and usability features of the PSBLAS

with Fortran 95 interface. The mathematical formulation of the algorithms is quite comparable to the PSBLAS implementation.

Efficiency improvements can be obtained from this basic implementation through optional parameters that are available in the subroutine interfaces.

Template from [7]	PSBLAS Implementation

Compute $r^{(0)} = b - Ax^{(0)}$	`call psb_geaxpby(one,b,zero,r,desc_a,info)`
	`call psb_spmm(-one,A,x,one,r,desc_a,info)`
	`bni = psb_geamax(b,desc_a,info)`
	`ani = psb_spnrmi(A,desc_a,info)`
	`rho = zero`

for $i = 1, 2, \ldots$
solve $Mz^{(i-1)} = r^{(i-1)}$
$\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
	`do  it = 1, itmax`
	`call psb_precaply(PR,r,z,desc_a,info)`
	`rho_old = rho`
	`rho = psb_gedot(r,z,desc_a,info)`

if $i = 1$
$p^{(1)} = z^{(0)}$
else
$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
endif
	`if (it == 1) then`
	`call psb_geaxpby(one,z,zero,p,desc_a,info)`
	`else`
	`beta = rho/rho_old`
	`call psb_geaxpby(one,z,beta,p,desc_a,info)`
	`endif`

$q^{(i)} = Ap^{(i)}$
$\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
	`call psb_spmm(one,A,p,zero,q,desc_a,info)`
	`sigma = psb_gedot(p,q,desc_a,info)`
	`alpha = rho/sigma`

$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	`call psb_geaxpby(alpha,p,one,x,desc_a,info)` `call psb_geaxpby(-alpha,q,one,r,desc_a,info)`

Check convergence:
$\lVert r^{(i)} \rVert_\infty \leq \epsilon(\lVert A \rVert_\infty \cdot \lVert x^{(i)} \rVert_\infty + \lVert b \rVert_\infty)$
	`rni = psb_geamax(r,desc_a,info)`
	`xni = psb_geamax(x,desc_a,info)`
	`err = rni/(ani*xni+bni)`
	`if (err.le.eps) return`

end	`enddo`

Table 2.1: Sample CG implementation

Template from [7]	PSBLAS Implementation

Compute $r^{(0)} = b - Ax^{(0)}$
 	
Choose q (e.g. $q = r^{(0)}$)
 	
for $i = 1, 2, \ldots$
$\rho_{i-1} = q^T r^{(i-1)}$
 	
if $i = 1$
$p^{(1)} = r^{(0)}$
else
if $\rho = 0$ **failure**
$\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
$p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1}v^{(i-1)})$
 	
endif
solve $M\hat{p} = p^{(i-1)}$
$v^{(i)} = A\hat{p}$
$\alpha_i = \rho_{i-1}/q^T v^{(i)}$
 	
$s = r^{(i-1)} - \alpha v^{(i)}$
 	
solve $M\hat{s} = s$
$t = A\hat{s}$
$\omega_i = t^T s / t^T t$
 	
$x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
 	
$r^{(i)} = s - \omega_i t$
 	
Check convergence:
$\lVert r^{(i)} \rVert_\infty \leq \epsilon(\lVert A \rVert_\infty \cdot \lVert x^{(i)} \rVert_\infty + \lVert b \rVert_\infty)$
 	
end
	`call psb_geaxpby(one,b,zero,r,desc_a,info)`
	`call psb_spmm(-one, A, x, one,r,desc_a,info)`
	`call psb_geaxpby(one,r,zero,q,desc_a,info)`
	`bni = psb_geamax(b,desc_a,info)`
	`ani = psb_spnrmi(A,desc_a,info)`
	`do  it = 1, itmax`
	`  rho_old = rho`
	`  rho = psb_gedot(q,r,desc_a,info)`
	`  if (it == 1) then`
	`  call psb_geaxpby(one,r,zero,p,desc_a,info)`
	`  else`
	`  if (rho==0) stop`
	`  beta = (rho/rho_old)(alpha/omega)`
	`  call psb_geaxpby(-omega,v,one,p,desc_a,info)`
	`  call psb_geaxpby(one,r,beta,p,desc_a,info)`
	`  endif`
	`  call psb_precaply(PR,p,phat,desc_a,info)`
	`  call psb_spmm(one,A,phat,zero,v,desc_a,info)`
	`  alpha = psb_gedot(q,v,desc_a,info)`
	`  alpha = rho/alpha`
	`  call psb_geaxpby(one,r,zero,s,desc_a,info)`
	`  call psb_geaxpby(-alpha,v,one,s,desc_a,info)`
	`  call psb_precaply(PR,s,shat,desc_a,info)`
	`  call psb_spmm(one,A,shat,zero,t,desc_a,info)`
	`  omega = psb_gedot(t,s,desc_a,info)`
	`  temp = psb_gedot(t,t,desc_a,info)`
	`  omega=omega/temp`
	`  call psb_geaxpby(alpha,phat,one,x,desc_a,info)`
	`  call psb_geaxpby(omega,shat,one,x,desc_a,info)`
	`  call psb_geaxpby(one,s,zero,r,desc_a,info)`
	`  call psb_geaxpby(-omega,t,one,r,desc_a,info)`
	
	`  rni = psb_geamax(r,desc_a,info)`
	`  xni = psb_geamax(x,desc_a,info)`
	`  err = rni/(ani*xni+bni)`
	`  if (err.le.eps) return`
	`enddo`

Table 2.2: Sample Bi-CGSTAB implementation

2.7 Preconditioners

Preconditioning is somehow regarded as "black magic". This is due to the fact that theory doesn't provide a reliable support in the choice of a preconditioner. It is clear that the influence of a preconditioning technique on the convergence behavior of an iterative method mostly depends on the characteristics of the system matrix and of the method itself. Anyway it is not possible a priori to say that one preconditioner is algebrically better than another and this perfectly explains the importance of providing a wide range of preconditioners techniques so that the user can find by itself which one is more suitable for his problem. Moreover, there are some other issues to consider when choosing a preconditioner such as balancing the overhead of building the preconditioner with the reduction in the number of iterations. PSBLAS contains the implementation of many preconditioning techniques some of which are very flexible thanks to the presence of many parameters that is possible to adjust to fit the user's needs:

- Diagonal Scaling
- Block Jacobi with ILU(0) factorization
- Additive Schwarz with the Restricted Additive Schwarz and Additive Schwarz with Harmonic extensions (see chapter 4)
- Two-Level Additive Schwarz; this is actually a family of preconditioners since there is the possibility to choose between many variants as explained in chapter 5

Preconditioners in PSBLAS are built and managed through the following subroutines.

psb_precset This subroutine defines which kind of preconditioner will be contained in the `p` preconditioner data structure (of the type described in 2.2.4). The type of preconditioner is specified by means of the `iv`, `rs` and `rv` arguments.

```
call psb_precset(p, ptype, iv, rs, rv, info)
```

psb_precbld It builds the preconditioner according to the rules fixed by the `psb_precset` subroutine. A detail of the actions perfomed inside this routine is given in chapters 4 and 5.

```
call psb_precbld(a, p, desc_a, info)
```

psb_precaply Applies the preconditioner. This routine is called inside the iterative method (see figures 2.1 and 2.2). Again, refer to chapters 4 and 5 for a description of the actions performed inside it.

```
call psb_precset(p, x, y, desc_a, info)
```

2.8 Error Handling

The PSBLAS library error handling policy has been completely rewritten in version 2.0. The idea behind the design of this new error handling strategy is to keep error messages on a stack allowing the user to trace back up to the point where the first error message has been generated. Every routine in the PSBLAS-2.0 library has, as last non-optional argument, an integer `info` variable; whenever, inside the routine, en error is detected, this variable is set to a value corresponding to a specific error code. Then this error code is also pushed on the error stack and then either control is returned to the caller routine or the execution is aborted, depending on the users choice. At the time when the execution is aborted, an error message is printed on standard output with a level of verbosity than can be chosen by the user. If the execution is not aborted, then, the caller routine checks the value returned in the `info` variable and, if not zero, an error condition is raised. This process continues on all the levels of nested calls until the level where the user decides to abort the program execution.

Figure 2.6 shows the layout of a generic `psb_foo` routine with respect to the PSBLAS-2.0 error handling policy. It is possible to see how, whenever an error condition is detected, the `info` variable is set to the corresponding error code which is, then, pushed on top of the stack by means of the `psb_errpush`. An error condition may be directly detected inside a routine or indirectly checking the error code returned returned by a called routine. Whenever an error is encountered, after it has been pushed on stack, the program execution skips to a point where the error condition is handled; the error condition is handled either by returning control to the caller routine or by calling the `psb\_error` routine which prints the content of the error stack and aborts the program execution.

Figure 2.7 reports a sample error message generated by the PSBLAS-2.0 library. This error has been generated by the fact that the user has chosen the invalid "FOO" storage format to represent the sparse matrix. From this error message it is possible to see that the error has been detected inside the `psb_cest` subroutine called by `psb_spasb` ... by process 0 (i.e. the root process).

```
subroutine psb_foo(some args, info)
   ...
   if(error detected) then
      info=errcode1
      call psb_errpush('psb_foo', errcode1)
      goto 9999
   end if
   ...
   call psb_bar(some args, info)
   if(info .ne. zero) then
      info=errcode2
      call psb_errpush('psb_foo', errcode2)
      goto 9999
   end if
   ...
9999 continue
   if (err_act .eq. act_abort) then
     call psb_error(icontxt)
     return
   else
     return
   end if

end subroutine psb_foo
```

Figure 2.6: The layout of a generic `psb_foo` routine with respect to PSBLAS-2.0 error handling policy.

In the following, a brief description of the error handling routines is given.

psb_errpush : this subroutine is used to push an error code on the error stack along with the name of the routine where the error has been detected. Other optional error informations can be specified depending on the particular error code.

call psb_errpush(err_code,name)

psb_error : this subroutine should be invoked when an error is detected to abort the program execution. It prints out the content of the stack (with a verbosity that can be set by the user) and then aborts the execution.

```
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: df_sample
Error from call to subroutine mat dist
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: mat_distv
Error from call to subroutine psb_spasb
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: psb_spasb
Error from call to subroutine psb_cest
============================================================
Process: 0.  PSBLAS Error (136) in subroutine: psb_cest
Format FOO is unknown
============================================================
Aborting...
```

Figure 2.7: A sample PSBLAS-2.0 error message. Process 0 detected an error condition inside the psb_cest subroutine

call psb_error(comm_context)

psb_set_err_verbosity : it can be used to set the verbosity of the error messages.

call psb_set_err_verbosity(verb_level)

psb_set_erraction : used to specify which action must be performed when an error condition is detected. The program execution can be interrupted in the same subroutine where the error has been detected or the control may be returned to the caller subroutine.

call psb_set_erraction(err_action)

psb_errcomm : it is a simple communication routine to notify other processes that an error has occured.

call psb_errcomm(comm_context,err_code)

Part II

Improving low level computing kernels

Chapter 3

Sparse Storage Formats

Contents

At the heart of the performance problem is that sparse operations are far more bandwidth-bound than dense ones. Most processors have a memory subsystem considerably slower than the processor, and this situation is not likely to improve substantially any time soon. Consequently, optimizations are needed, likely to be intricate, and very much dependent on architectural variations even between closely related versions of the same processor.

The classical approach to the optimization problem consists in hand tuning the software according to the characteristics of the particular architecture which is going to be used, and according to the expected characteristics of the data. This approach yields significant results but poses a serious problem on portability because the software becomes tightly coupled with the underlying architecture.

The Self Adaptive Numerical Software efforts [25, 81] aim to address this problem. The main idea behind this new approach to numerical software

optimization consists in developing software that is able to adapt its characteristics according to the properties of the underlying hardware and of the input data.

The state of kernel optimization in numerical linear algebra is furthest advanced in dense linear algebra. The ATLAS software [81], for example, gives near optimal performance on the BLAS kernels. Factorizations of sparse matrices (MUMPS [2, 84], SuperLU [51], UMFPACK [23, 86]) also perform fairly well, since these lead to gradually denser matrices throughout the factorization. Kernel optimization leaves most to be desired in the optimization of the components of iterative solvers for sparse systems: the sparse matrix-vector product and the sparse ILU solution.

In this chapter the theory and the implementation of an adaptive strategy for sparse matrix-vector products is presented. The optimization studied consists in performing the operation by blocks instead by single entries, which allows for more optimizations, thus possibly leading to faster performance than the scalar – reference – implementation. The parameter optimized is the choice of the block dimensions, which is a function of the particular matrix and the machine.

An approach along these lines has already been studied in [48, 79] and, more recently, extended in [78]. Essentially the same optimizations is employied, but relaxing one restriction in that research. However, a more accurate performance model is presented, which leads to better predictions of the block size, and consequently higher performance. The accuracy of the models and the resulting performance numbers will be also compared.

Other authors have proposed similar and different techniques for accelerating the sparse matrix-vector product. For instance, Toledo ([71] and the references therein) mentions the possibility of reordering the matrix (in particular with a bandwidth reducing algorithm) to reduce cache misses on the input vector. Pinar and Heath [60] also consider reordering the matrix; they use it explicitly to find larger blocks, which leads to a Traveling Salesman Problem.

While the reordering approach gives an undoubted improvement, there are two reasons for not considering it in this study. For one, in the context of a numerical library for sparse kernels, permuting the kernel operations has many implications for the calling environment. Secondly, the blocking strategy presented hereby can equally well be applied to already permuted matrices, so the following discussion will be orthogonal to this technique.

Blocking approaches have also been tried before. Both Toledo [71] and Vuduc [79] propose a solution where a matrix is stored as a sum of differently blocked matrices, for instance on with the 2×2 blocks, one with 2×1 blocks, and the third one with the remaining elements. The code described here has been released as a package 'AcCELS' (Accelerated Compressed-storage Elements for Linear Solvers); it will also be part of the PSBLAS library [39].

In addition to the matrix-vector product, also a block-optimized version of the triangular solve operation is include in the software package. This routine is useful in direct solution methods (for the backward and forward solve) and in the application of some preconditioners.

Matrix-vector multiplication and triangular system solving are very common yet expensive operations in sparse algebra computations. total time These two operations typically account for more than 50% of the total time spent in the solution of a linear sparse system using an iterative method and, moreover, they tend to perform very poorly on modern architectures. There are several reasons for the low performance of these two operations:

- **Indirect addressing/low spatial locality:** sparse matrices are stored in data structures where in addition to the values of the entries the row indices and the column indices have to be explicitly stored. The most common formats are Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) storage [7, §4.3]. During the matrix-vector product, in the case of CSR storage of the matrix (resp. CSC) the discontinuous way the elements of the source vector (resp. destination vector) are accessed is a bottleneck that causes low spatial locality.

- **Low temporal locality:** In order to minimize memory access, it is important to maximize the number a data is reused. During a sparse matrix-vector product with a matrix stored in Compressed Sparse Row (CSR) format, the elements of the matrix are accessed sequentially in row order and are used once, the elements of the destination vector are accessed sequentially and each of them is reused as many times as the number of elements in the corresponding row of the sparse matrix which is optimal with respect to the temporal locality. Unfortunately, the elements of the initial vector are accessed according to the column indices of the elements of the active row of A. The elements of x are reused during the matrix-vector product when their row indices belongs to two (or more) consecutive rows of the matrix A where there are elements on the corresponding column. Using the CSR storage format for the matrix implies that all the computations are performed row by row, thus, while moving from one row to the next the cache is in general overwritten. This leads to poor temporal locality of the source vector.

- **Low ratio between floating-point operations and memory operations:** apart from the elements of the matrix, the indices also have to be explicitly read from memory which leads to a high consumption of the

CPU-memory bandwidth. Basically, there are two reads per floating-point multiply-add operation. The ratio is one in the dense case. The comparison with the dense linear algebra is even starker considering that there is one write operation per row. Since there typically are far fewer elements per row in the sparse case, this type of overhead is relatively higher in the sparse case. Moreover retrieving and manipulating the column/row indices information implies an amount of integer operations that is not negligible.

- **High loop overhead:** Connected to the low number of elements per row in sparse systems, the loop overhead is correspondingly higher. Furthermore, since the loop length is not constant throughout the matrix, there is more indexing computation involved, and because of the non-uniformity several compiler techniques such as loop interchange are not possible in straightforward manner. Consider the matrix-vector product $Ax = y$ where A is a sparse matrix stored in Compressed Sparse Row (CSR) format [7] and x and y are dense vectors. All the elements of the matrix A are accessed sequentially in row order and cannot be reused; all the elements of the destination vector y are accessed sequentially but each of them can be reused as many times as the number of elements in the corresponding row of A; finally the elements of x are accessed according to the column indices of the elements in each row of A and are not reused. The elements of x can be reused only if in two (or more) consecutive rows of the matrix A there are elements on the same column; unfortunately using the CSR storage format for A means that all the computations are performed row by row, thus, while moving from one row to the next the cache can be completely overwritten.

The optimization of the sparse matrix-vector operations presented here consists in *tiling* the matrix with small dense blocks that are chosen to cover the nonzero structure of the matrix. This causes an improvement in scalar performance due to reduced indexing and greater data locality of the dense blocks. Unfortunately the number of operations increases due to the operations performed on the zeros arising in the dense blocks (this phenomenon will be referred to as *fill-in*). There is clearly a trade-off, which is going to be analyzed in sections 3.2 and 3.3.

Optimizing the sparse matrix-vector product kernels has two components:

1. Assessing the performance for blocks of different sizes. This performance is a non-trivial function of various architectural features;

2. Finding the best tiling for a given matrix. Each different block size results in a different number of stored nonzeroes, and therefore a different number of operations performed. This needs to be balanced with the performance of the dense blocks, in a way that will be explained below.

Previously, the Atlas project [81] has been singularly successful in optimizing dense linear algebra kernels. The ATLAS strategy consists in optimizing the different parameters to the architecture in a installation phase.

In the sparse case, the structure of the matrix has a great influence on the optimal parameters and the resulting performance. A static approach like this one is not possible. Since the structure of the matrix is only known at run time, the choice of the parameters for the sparse matrix-vector product is performed at run time. Consequently the block size selection is on the basis of information that is gathered in two distinct phases:

1. **Installation-time phase:** in this phase the impact of the architecture characteristics on the performance of the block operations is analyzed.

2. **Run-time phase:** in this phase the sparsity structure of the matrix is analyzed to understand how it influences the fill-in ratio.

3.1 Storage Formats Overview

This section describes some of the most widely used storage format for sparse matrices. Many more formats are described in literature (see [7, 18]) but many of them have similar properties from a performance point of view and thus it's not worth describing or implementing all of them. In the following the same notation as in [18] will be used for the description of the storage formats:

$NNZ(A)$ The number of entries of a matrix A where an entry means any matrix coefficient which is handled explicitly. Typically the entries of A are simply the nonzero elements of A. If the entries are $r \times c$ blocks, then $NNZ(A)$ is the number of block entries.

$NZE(A)$ The set of entries of A.

$NZI(A)$ The set of row indices corresponding to the entries of matrix A. The ordering of the elements in $NZI(A)$ should be the same as those of $NZE(A)$.

$NZJ(A)$ The set of column indices corresponding to the entries of matrix A. The ordering of the elements in $NZI(A)$ should be the same as those of $NZE(A)$.

A_{i*} The vector consisting of the elements of the i-th row of A.

A_{*j} The vector consisting of the elements of the j-th column of A.

$diag_i(A)$ The vector consisting of the elements from the i-th diagonal of A.

The matrix in 3.1 will be used to show examples of each storage scheme.

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 \\ 0 & 0 & a_{23} & a_{24} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 \\ a_{51} & a_{52} & 0 & 0 & a_{55} \end{pmatrix} \tag{3.1}$$

3.1.1 The COO storage Format

In the COO storage format each nonzero element is stored along with its row and column indices. Three arrays are thus needed to store a matrix in this format:

VAL An array of length $NNZ(A)$ that contains the elements of $NZE(A)$ in any order.

INDX An array of length $NNZ(A)$ that contains the elements of $NZI(A)$ with the same ordering as in VAL.

JNDX An array of length $NNZ(A)$ that contains the elements of $NZJ(A)$ with the same ordering as in VAL.

The matrix A in 3.1 can thus be represented as:

$$\begin{aligned} \text{VAL} &= (\ a_{11}\ a_{13}\ a_{14}\ a_{23}\ a_{24}\ a_{31}\ a_{32}\ a_{33}\ a_{34}\ a_{42}\ a_{44}\ a_{51}\ a_{52}\ a_{55}\ ), \\ \text{INDX} &= (\ 1\ 1\ 1\ 2\ 2\ 3\ 3\ 3\ 3\ 4\ 4\ 5\ 5\ 5\ ), \\ \text{JNDX} &= (\ 1\ 3\ 4\ 3\ 4\ 1\ 2\ 3\ 4\ 2\ 4\ 1\ 2\ 5\ ). \end{aligned}$$

This storage format is very easy and almost straightforward to implement. Anyway it is not very good for performance. As stated before, elements must not be stored in a specific order (provided that in the three arrays the ordering remains the same) but it must be noted that row or column ordering can improve performance thanks to a more regular memory access pattern.

3.1.2 The CSR storage Format

In the Compressed Sparse Row storage format each nonzero element is stored along with its column index sorted in a row-wise order. Offset values are then used to access each row individually. As for the COO format, three vectors are used to represent a matrix in CSR form:

VAL An array of length $NNZ(A)$ that contains the elements of $NZE(A)$ in row-wise order.

$$VAL = (NZE(A_{1*}, NZE(A_{2*}, ..., NZE(A_{m*}))$$

JNDX An array of length $NNZ(A)$ that contains the elements of $NZJ(A)$ with the same ordering as in VAL.

$$JNDX = (NZJ(A_{1*}, NZJ(A_{2*}, ..., NZJ(A_{m*}))$$

INDX An array of length $m+1$ where $INDX(i)$ contains a pointer to the beginning of the i-th row inside the previous two arrays. $INDX(m+1) = NNZ(A)+1$. Thus $INDX(i)$ points to the location in VAL where the first element of $NZE(A_{1*})$ is stored.

$$INDX(i) = \sum_{j=1}^{i-1} NNZ(A_{j*}) + 1 \qquad i = 1, ..., m+1$$

The matrix A in 3.1 can thus be represented as:

VAL = (a_{11} a_{13} a_{14} a_{23} a_{24} a_{31} a_{32} a_{33} a_{34} a_{42} a_{44} a_{51} a_{52} a_{55}),
JNDX = (1 3 4 3 4 1 2 3 4 2 4 1 2 5),
INDX = (1 4 6 10 12 15).

Using the offset information contained in $INDX$ implies an heavy usage of indirect addressing but anyway this storage format has proven to be efficient on many architectures and this is the reason why it is probably the best known and most widely used among all the sparse storage formats.

3.1.3 The JAD storage Format

Standard JAD

The Jagged Diagonal storage format requires a permutation matrix P that sorts the rows of the matrix A in descending order of $NNZ(A_{i*})$. Such a permutation matrix can be represented by an integer vector $IPERM$. Let $\bar{A} = PA$ be the permuted matrix and i' denote the row of $\bar{A}$ corresponding to the i-th row of A, then $i = IPERM(i')$. For this storage format four arrays are needed:

VAL An array of length $NNZ(A)$ that contains the elements of A. The first element in this array is the first element of $NZE(\bar{A}_{1*}$ followed by the first element of $NZE(\bar{A}_{2*}$ and so on up to the first value of $NZE(\bar{A}_{m*}$. The $m+1$-th element of VAL is the second element of $NZE(\bar{A}_{1*}$ and so on.

JNDX An array of length $NNZ(A)$ that contains the elements of $NZJ(A)$ with the same ordering as in VAL.

PNTR An integer array of length $MAXNZ + 1$ where

$$MAXNZ = \max_{1 \le i \le m} NNZ(A_{i*})$$

such that $PNTR(j)$ points to the location in VAL of the j-th element of $NZE(\bar{A}_{1*})$. $PNTR(MAXNZ + 1)$ is set to $NNZ(A) + 1$.

IPERM An integer array of length m such that $i = IPERM(i')$.

A suitable permutation matrix for matrix 3.1 would be:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

and thus the permuted matrix $\bar{A}$ would be:

$$A = \begin{pmatrix} a_{31} & a_{32} & a_{33} & a_{34} & 0 \\ a_{11} & 0 & a_{13} & a_{14} & 0 \\ a_{51} & a_{52} & 0 & 0 & a_{55} \\ 0 & 0 & a_{23} & a_{24} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 \end{pmatrix}$$

The JAD representation for matrix 3.1 is:

VAL = (a_{31} a_{11} a_{51} a_{23} a_{42} a_{32} a_{13} a_{52} a_{24} a_{44} a_{33} a_{14} a_{55} a_{34}),
JNDX = (1 1 1 3 2 2 3 2 4 4 3 4 5 4),
INDX = (1 6 11 14 15),
PNTR = (3 1 5 4 4).

The Jagged Diagonal storage format is almost difficult to implement but has proven to be efficient on vector machines.

JAD Implementation in PSBLAS

The PSBLAS library uses a minor variant of the Jagged Diagonal storage format that can be considerably more efficient especially with those matrices that have a regular sparsity structure.

Matrices represented in this format are partitioned into sets of rows. All the rows that belong to the same set have the same number of nonzero

elements. Where needed "short" rows can be padded with zero values to be included in block. This padding zero values are called "fill-in" elements and within each block the number of these elements cannot be greater than an upper bound defined by a tolerance value. Whenever the fill-in ratio becomes higher than the given tolerance, the CSR storage format is used to represent the tail of the block. This tail includes all the elements whose column index is greater than the column index of the last element in the shortest row of the block.

The data structure used to represent a matrix in this JAD variant has the following records:

NG A scalar value that defines the number of blocks into which the matrix is partitioned.

VAL An array of length $NNZ(A) + fill - in$ containing the nonzero values of A $NZE(A)$ plus the eventual fill-in elements that have been added. All the elements in a block are stored close together in a column-wise order.

JNDX An array of length $NNZ(A)+fill-in$ containing the column indices for the values in VAL

INDX A bi-dimensional array of size $3 \times NG$. $INDX(1, i)$ contains the row index of the first row in the i-th block of $\bar{A}$. $INDX(2, i)$ points to the element of $PNTR$ containing the position of the first column of the i-th block within VAL and $JNDX$. $INDX(3, i)$ points to the element of $PNTR$ containing the position of the first column of the CSR tail of the i-th block within VAL and $JNDX$.

PNTR An array containing pointers to the beginning of each column of each block inside VAL and $JNDX$.

Figure 3.1 shows the portion of a block that is stored as a CSR tail.

3.2 The Block Sparse Matrix Format

The Block Compressed Sparse Row storage format for sparse matrices exploits the benefits of data blocking in numerical computations. This format is similar to the CSR format except that single value elements are replaced by dense blocks of general dimensions $r \times c$. Thus a BCSR format with parameters $r = 1$ and $c = 1$ is equivalent to the CSR format. All the blocks are row-aligned which implies that the first element of each block (i.e., the

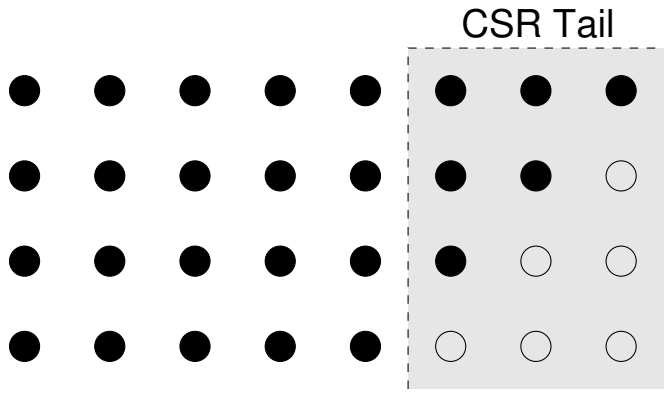

Figure 3.1: The CSR tail of a JAD block

upper leftmost element) has a global row index that is a multiple of the block row dimension r. It is possible to choose whether or not to let the blocks also be column-aligned.

A matrix in BCSR format is thus stored as three vectors: one that contains the dense blocks (whose elements can be stored by row or by column); one that contains the column index of each block (namely the column index of the first element of each block); and one which contains the pointers to the beginning of each block-row inside the other two vectors (a block row is a row formed by blocks, i.e. an aligned set of r consecutive rows).

Formally (in Fortran 1-based indexing),

for j=`ptr`[i]...ptr[i+1]-1:
    for k=1...(r*c):
        `elem`[(j-1)*r*c+k] contains
        $A((i-1)*r+(k-1)/c+1, \texttt{col\_ind}[j]+\mathrm{mod}(k-1,c)+1)$

All elements of the matrix A belong to a small dense block; this means that when the number of nonzero elements is not enough to build up a block, zero values are explicitly stored to fill the empty spaces left in the blocks. These added zero values are called fill-in elements (see red-plus elements on figure 3.2 right).

The fill-in ratio is computed as the ratio between the total number of elements (original nonzeroes plus fill-in zeros) and the nonzero elements; for the matrix in Figure 3.2(right) with 3×3 block size the fill-in ratio is 1.59. Performing the matrix-vector product with the matrix in Figure 3.2(right) stored in BCSR format with 3×3 block size, 1.59 times as many floating point operations as in the case of the CSR format have to be executed.

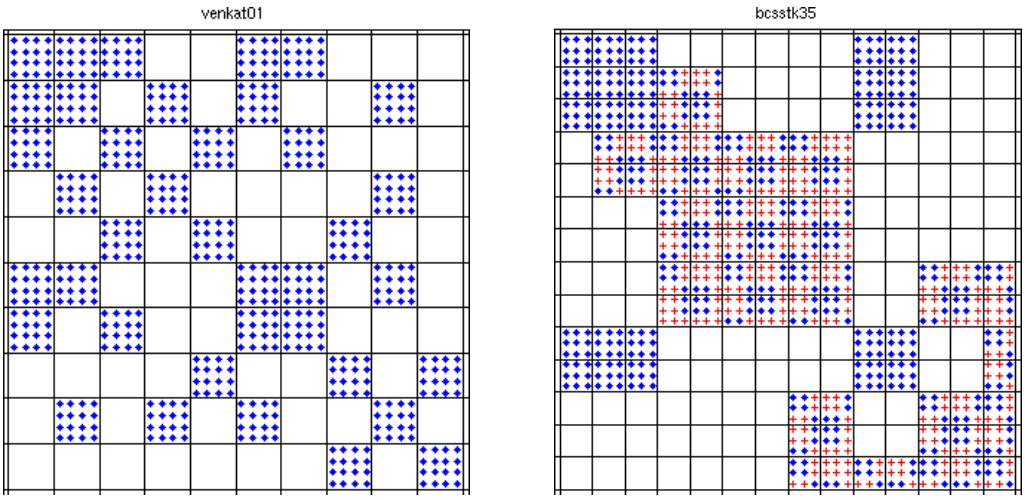

Figure 3.2: Blocking of a portion of venkat01 matrix with 4×4 blocks (left) and bcsstk35 matrix with 3×3 blocks (right).

Fortunately, in most sparse matrices the elements are not randomly distributed, so such a block tiling often makes sense. Either the matrices have an intrinsic block structure (in which case the fill-in is zero) like in the case of the matrix in figure 3.2 (left), or elements are sufficiently clustered so that it is possible to find a block size for which the fill-in is low. However, experimental results presented in the next sessions show that, even in the cases where the fill-in ratio is high on the average, a reduction in the matrix-vector product execution time can still be achieved thanks to the substantial increase in the flop rate.

A lower fill-in ratio can be often achieved by relaxing the limitation that the blocks must be column aligned. Each block inside a block row begins at a column index that is not necessarily a multiple of the column size c. While this choice increases the time spent during the matrix building phase since more possibilities have to be evaluated, it has no extra overhead during the matrix-vector product operation.

Figure 3.3 shows the tiling of the same matrix with 3×3 row and column aligned blocks on the left and row aligned but column unaligned blocks on the right. In this case the fill-in ratio is reduced form 2.83 to 2.36.

For the matrices in the testset described in appendix A, the fill in ratio for unaligned and aligned blocks is shown in figure 3.4 Using column-unaligned blocks yields a substantial reduction in fill-in ratio especially as the size of the small dense blocks grows and thus a reduction in the matrix-vector product execution times. Anyway having column-unaligned blocks raise the need for a more complex algorithm for the matrix assembly and also for the fill-in

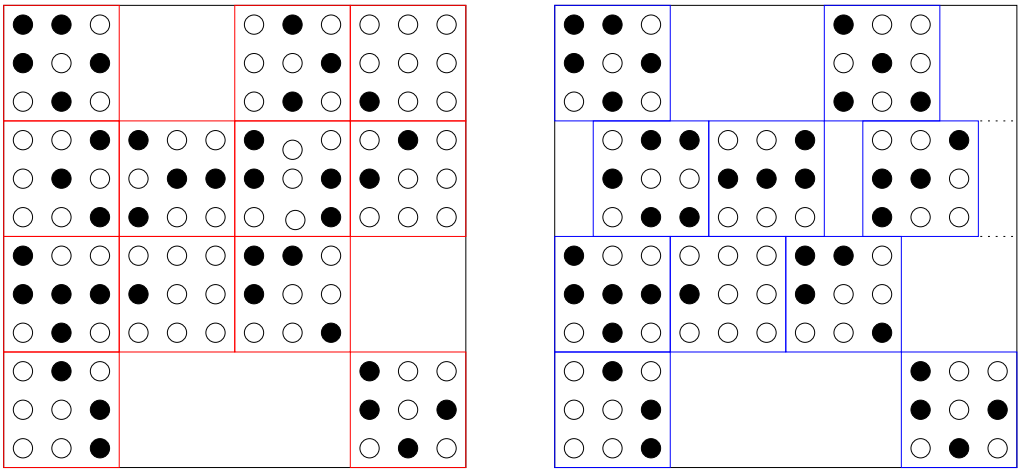

Figure 3.3: Fill-in for 3×3 row and column aligned blocks (left) and row aligned but column unaligned blocks (right).

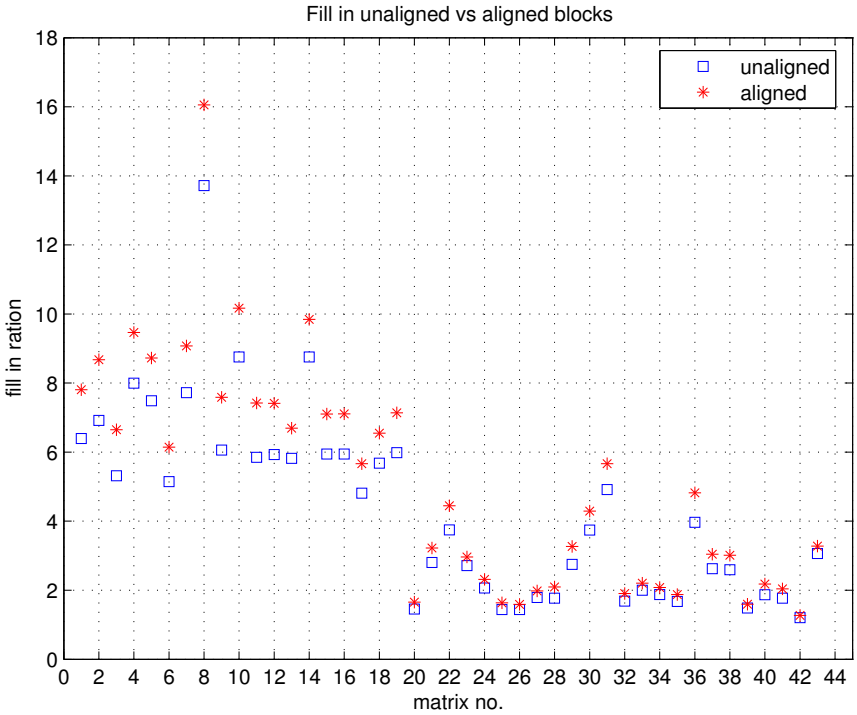

Figure 3.4: Fill in ratio for matrices in testset with aligned and unaligned blocks.

estimate. Depending on the matrix, the architecture and the user's needs it could be valuable to choose column aligned blocks instead of unaligned ones.

Figures 3.5 3.6 3.7 3.8 show how expensive is to estimate the fill-in and convert a matrix to the BCSR storage format with unaligned blocks as compared with the case where aligned blocks are used (the top part of each figure). It is possible to see that the average cost of estimating the fill-in (with accuracy set to 1%) and converting a matrix to the BCSR storage format having unaligned blocks can be bound between one and five times the cost of these operations in the case of aligned blocks. Evaluating the benefits of the fill-in reduction due to the unaligned blocks (as it can be done referring to figure 3.4) it is possible to understand when using unaligned blocks yields an effective advantage with respect to having aligned blocks. On the bottom part of each figure 3.5 3.6 3.7 3.8 the cost of estimating the fill-in (with accuracy set to 1%) and converting a matrix into the BCSR storage format is shown, measured in numbers of unblocked matrix-vector product. Depending on the architecture (where "architecture" means hardware plus compiler) and the particular matrix used, the cost of estimating the fill-in varies from few tens up to more than one hundred (on the Itanium2 architecture 3.7). Thus, the BCSR format results to be valuable only when used in those applications where a high number of matrix-vector products must be performed like, for example, the solution of systems through iterative solvers.

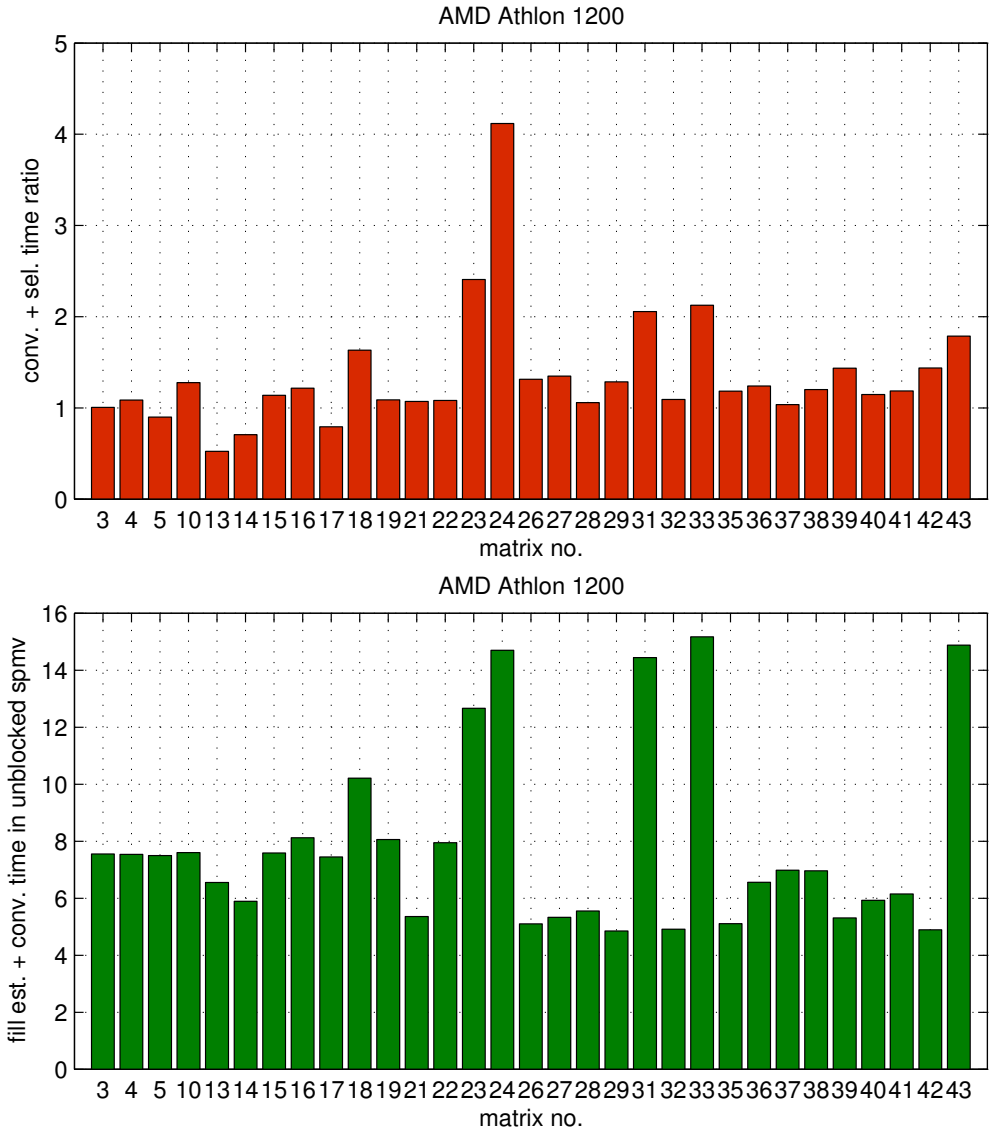

Figure 3.5: Comparison between the time spent in fill-in estimate (accuracy=0.01) plus storage format conversion for aligned and unaligned blocks on the AMD Athlon 1200 machine (*top*). Cost of fill-in estimate measured in unblocked matrix-vector products on the AMD Athlon 1200 machine (*bottom*).

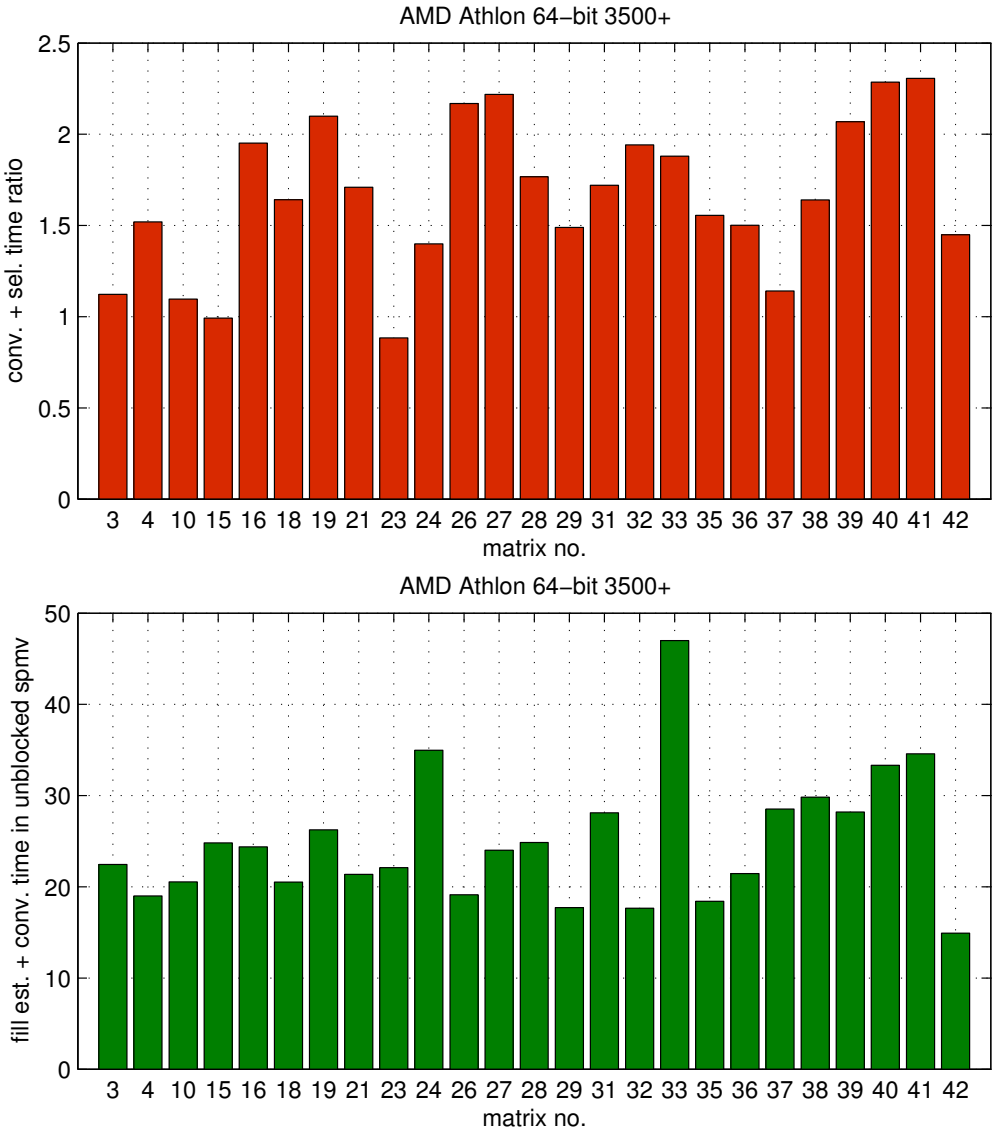

Figure 3.6: Comparison between the time spent in fill-in estimate (accuracy=0.01) plus storage format conversion for aligned and unaligned blocks on the AMD Athlon 64-bit 3500+ machine (*top*). Cost of fill-in estimate measured in unblocked matrix-vector products on the AMD Athlon 64-bit 3500+ machine (*bottom*).

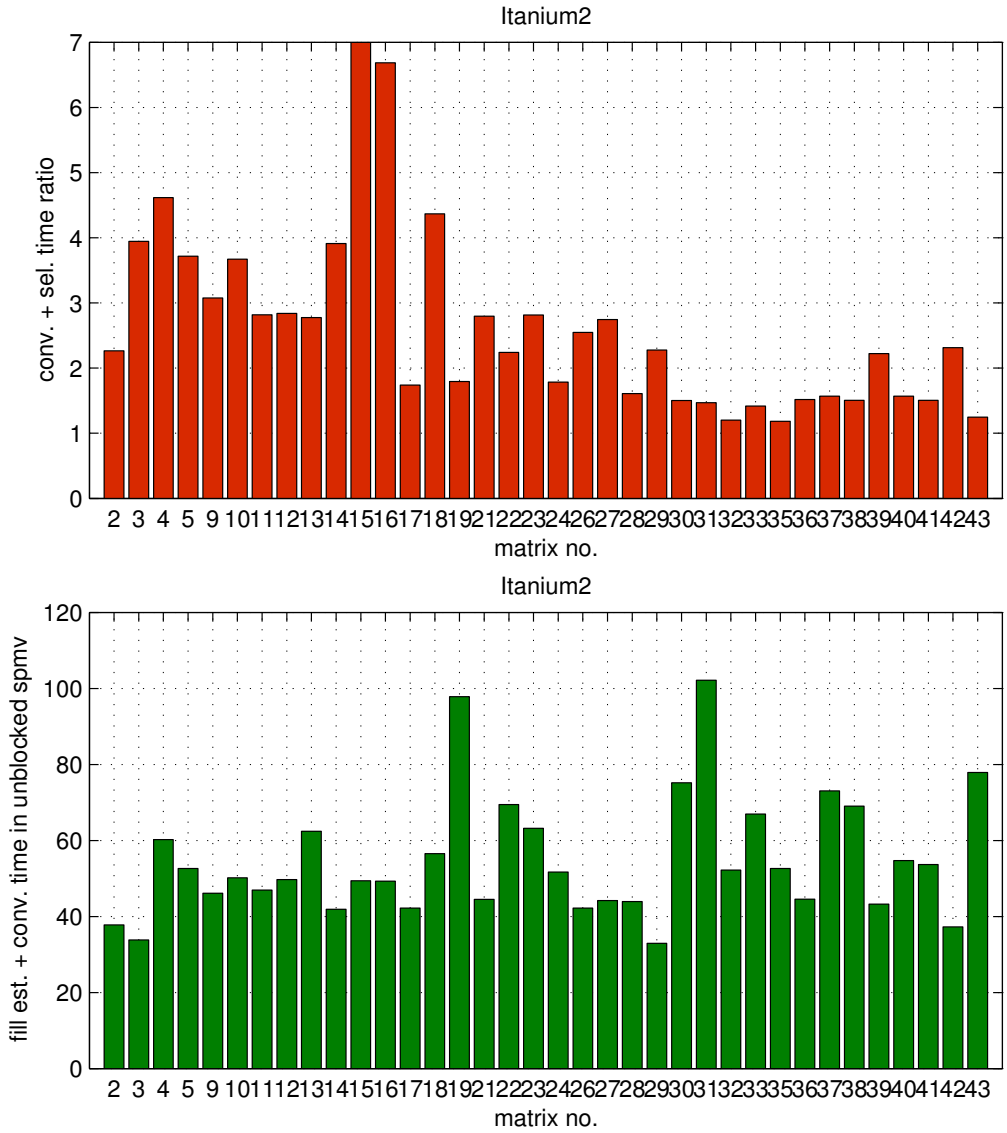

Figure 3.7: Comparison between the time spent in fill-in estimate (accuracy=0.01) plus storage format conversion for aligned and unaligned blocks on the Itanium2 machine (*top*). Cost of fill-in estimate measured in unblocked matrix-vector products on the Itanium2 machine (*bottom*).

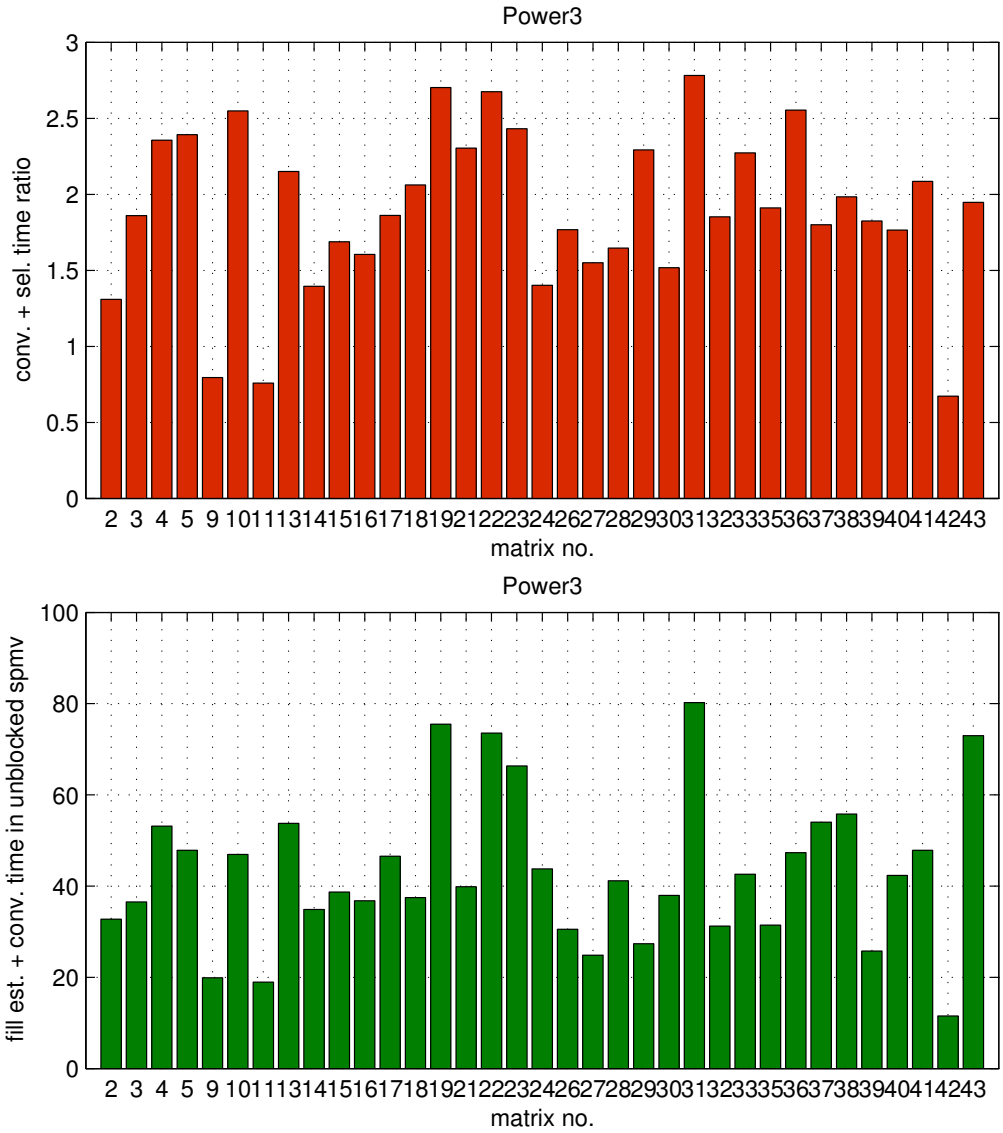

Figure 3.8: Comparison between the time spent in fill-in estimate (accuracy=0.01) plus storage format conversion for aligned and unaligned blocks on the Power3 machine (*top*). Cost of fill-in estimate measured in unblocked matrix-vector products on the Power3 machine (*bottom*).

3.3 Performance Optimization and Modeling

The BCSR Matrix-Vector Product source code

Figures 3.9 and 3.10 show the source code for the matrix-vector product $y = Ax$ for the BCSR (in the case of 2×3 blocks) and for the reference CSR (equivalent to the 1×1 BCSR) formats. Array `ia1` contains the column indices for the (block) elements, array `ia2` contains the (block-row) row pointers and array `aspk` contains the nonzero (blocks) elements.

```
...
for(i=0;i<*m;i++,y+=2){
    int j;
    register double y0=y[0];
    register double y1=y[1];
    for(j=ia2[i];j<ia2[i+1];j++,ia1++,aspk+=6){
        y0 += aspk[0]*x[*ia1 +0];
        y1 += aspk[3]*x[*ia1 +0];
        y0 += aspk[1]*x[*ia1 +1];
        y1 += aspk[4]*x[*ia1 +1];
        y0 += aspk[2]*x[*ia1 +2];
        y1 += aspk[5]*x[*ia1 +2];
    }
    y[0]=y0;
    y[1]=y1;
}
...
```

Figure 3.9: Matrix-vector product source code for 2×3 BCSR storage format.

The difference of performance between the two implementations for the matrix-vector product may be understood from the following analysis:

- memory space occupancy, and thus also memory-CPU bandwidth consumption may be eventually reduced. Following the discussion presented in [79] it is easily possible to quantify to which extent the BCSR format affects memory space occupancy and memory-CPU bandwidth consumption. Let k be the number of nonzeroes in A, and let K_{rc} be the number of nonzero $r \times c$ blocks in the BCSR representation of A. The fill-in ratio if equal to $f_{rc} = \frac{K_{rc} \cdot r \cdot c}{k}$. The matrix, as represented in BCSR format, has

```
...
for(i=0;i<*m;i++,y+=1){
  register double y0=y[0];
  for(j=ia2[i];j<ia2[i+1];j++,ia1++,aspk+=1){
    y0 += aspk[0]*x[*ia1+0];
  }
  y[0]=y[0];
}
...
```

Figure 3.10: Matrix vector product source code for reference CSR implementation.

$M_{rc} = \lceil \frac{m}{r} \rceil$ block-rows (where m is the number of rows of the matrix). Thus to represent a matrix in BCSR format a double-precision array as big as $K_{rc} \cdot r \cdot c$, an integer array of dimension K_{rc} to store the column indices of each block end another integer array of dimension $M_{rc} + 1$ to store the row pointers are needed. Assume that there are γ integers in a floating point word (which, for example, means $\gamma = 2$ with 32-bits integer values and 64-bits floating point values). The memory occupancy for a matrix in the BCSR format is

$$\begin{aligned} V_{rc} &= K_{rc} \cdot r \cdot c + \frac{1}{\gamma} K_{rc} + \frac{1}{\gamma} \left(\left\lceil \frac{m}{r} \right\rceil + 1 \right) \\ &= k f_{rc} \left(1 + \frac{1}{\gamma r c} \right) + \frac{1}{\gamma} \left(\left\lceil \frac{m}{r} \right\rceil + 1 \right) \end{aligned} \tag{3.2}$$

To understand the difference in memory occupancy between the CSR and the BCSR formats assume that $k >> m$ (which is the case for most of the sparse matrices coming from real world applications) and that $\gamma = 2$. These assumptions imply:

$$\frac{V_{11}}{V_{rc}} \approx \frac{\frac{3}{2} k}{k f_{rc} \left(1 + \frac{1}{2rc}\right)} = \frac{3}{2} \cdot \frac{1}{f_{rc} \left(1 + \frac{1}{2rc}\right)} \tag{3.3}$$

This means that in the optimistic case where there is no fill-in for the BCSR representation, it is possible to expect a reduction in the memory occupancy that is no more than 3/2 with respect to the CSR case. This translates in the fact that whenever the fill-in ratio is bigger than 3/2 more memory space is required to store a matrix in BCSR than in CSR.

- the inner loop, i.e. the loop where all the computations related to a small dense block are performed, is completely unrolled. This reduces loop overhead in the overall matrix-vector product computation and offers the opportunity to have a better scheduling of the instruction which results in a better usage of the processor pipeline.

- register reuse is substantially increased. Elements of the source vector x accessed within the inner loop can be held in registers and reused as many times as the column block dimension c. As the number of registers is limited and usually almost low, beyond a certain block size, performance is likely to be degraded by register spilling, as also pointed out in [79]). Register spilling is a phenomenon that happens when a segment of code tries to use more variables than the available number of registers; this means that variables have to be ,migrated back and forth from memory.

- memory access pattern is more regular as the $r \times c$ size of the dense blocks increases. Like in the dense matrix-vector product memory is accessed with a fixed stride when dealing with each single dense block which yields better cache behavior.

Architecture	ref. (MFlops)	best (MFlops)	speedup	`dgemv` (MFlops)
Itanium2	343.4	1482.6	4.31	1440.4
Athlon AMD 64-bit 3500+	452.0	732.5	1.62	750.8
Xeon	288.8	459.6	1.59	403.2
Athlon AMD 1800	105.7	209.8	1.98	206.6
Athlon AMD 1200	65.0	265.4	4.08	262.2
Power3	113.9	194.6	1.70	193.0
PentiumIII	49.1	142.0	2.89	150.2
MIPS	42.1	89.9	2.13	95.0

Table 3.1: This table shows a comparison between Flop rates of the reference implementation and the best case BCSR (i.e. the block size which yields the highest rate). The absolute speedup is reported on the third column while the dense `dgemv` (ATLAS library implementation) flop rate is reported on the fourth column.

To evaluate the performance of the source code listed in figure 3.9 for each different combination of block sizes r and c up to a certain limit, a

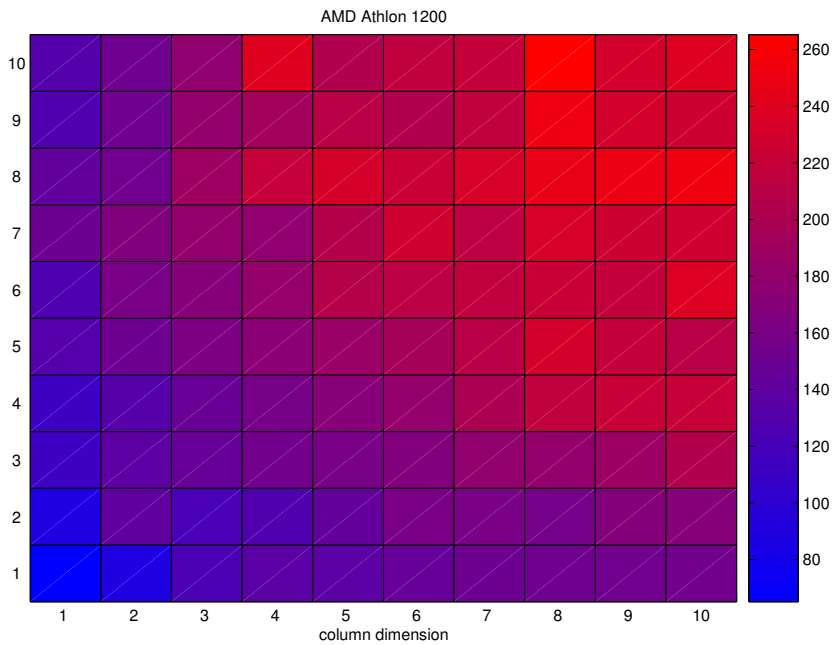

Figure 3.11: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an AMD Athlon 1200 architecture.

1500×1500 dense matrix stored in BCSR format has been used. This choice (also suggested by [79]) is almost natural since the storage of a dense matrix in BCSR format is not affected by any fill-in (when blocks dimension is not a submultiple of the matrix dimension there is some fill-in but it is very small and thus negligible) and then the attention can be focused on just flop rates without taking care of "noise" introduced by the presence of padding zeroes. The block size has been limited to 10×10 because beyond this limit usually the fill-in ratio for matrices coming from real-world applications is unbearable.

Figures 3.11 to 3.15 show the flop rate of the matrix-vector product operation on different architectures for the 1500×1500 dense matrix stored in BCSR format with block sizes between 1×1 and 10×10. The columns on the right side of each figure show that blue squares denote low values and red squares denote high values for flop rates.

Looking at figures 3.11 to 3.15 the following information can be derived:

- performance is a very irregular function of the block dimensions r and c. As pointed out before, performance should grow with higher block sizes thanks to an higher register reuse (achieved with higher values for r)

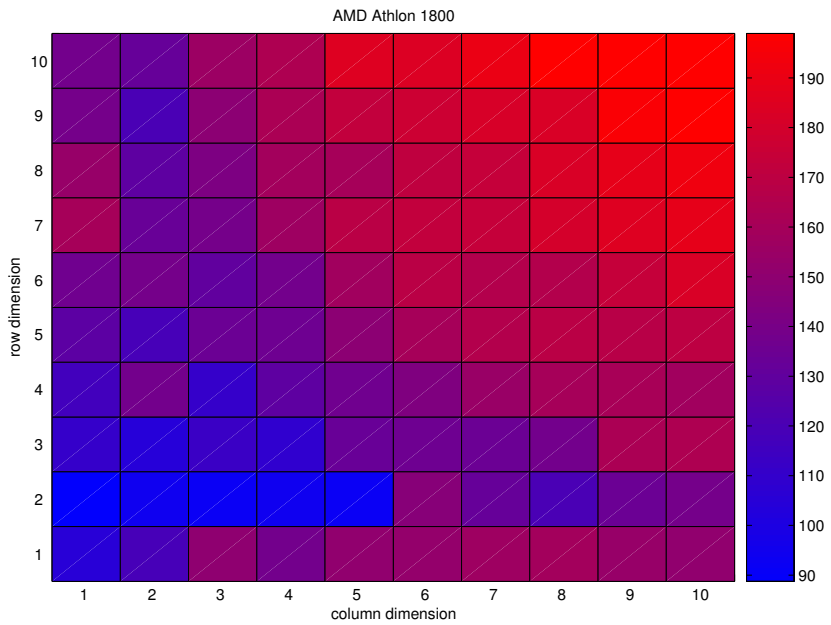

Figure 3.12: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an AMD Athlon 1800 architecture.

and a better memory access pattern (achieved with higher values for c). This behavior is only partially observed in these tests: the performance of the matrix-vector product operation in BCSR format on the AMD Athlon 1200 (figure 3.11), AMD Athlon 1800 (figure 3.12) and MIPS (figure 3.15) architectures seems to have a regular growth with higher values for the block sizes but figures 3.14 3.13 3.18 3.16 3.17 don't show any relation between flop rates and block dimension.

- register spilling doesn't seem, in general, to affect flop rates for higher dimensions of the dense blocks. Only on the Power3 architecture 3.17 it is possible to detect a degradation of performance for higher dimensions of the blocks that is likely to be triggered by register spilling.

- performance behavior is deeply affected by architecture characteristics and it is not possible, in general, to derive a general model that describes how performance changes according to different values of r and c.

- performance behavior is also deeply affected by compilers. Figure 3.14 shows that two different compilers generate code whose flop rate changes differently with the block dimensions r and c.

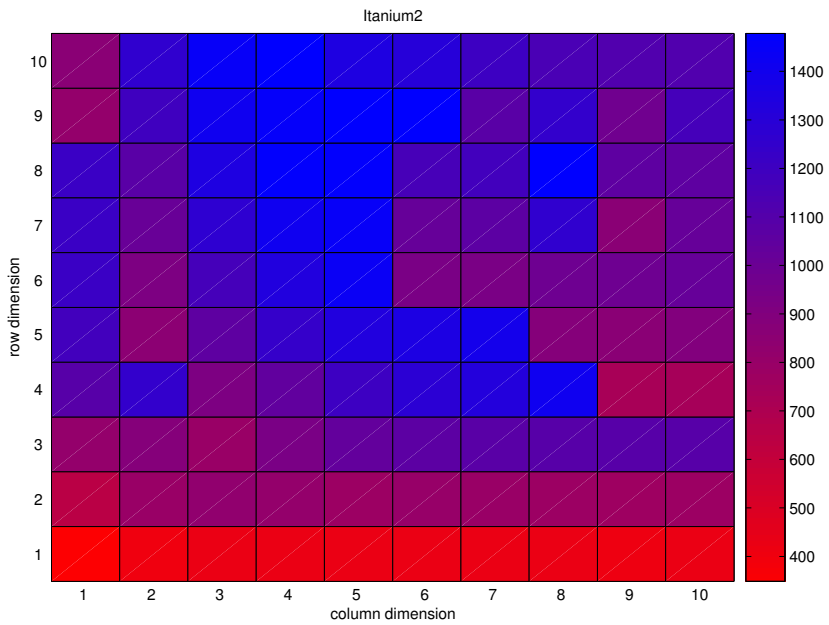

Figure 3.13: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an Itanium2 architecture.

- speedup can be considerable going from a minimum of 1.59 on the Xeon architecture up to a maximum of 4.31 on the Itanium2 one although we don't have, at the moment, a precise understanding of all the factors that contribute to it.

- the performance achieved with the best combination of r and c on most of the architectures used is higher than the the performance of the dense matrix-vector product operation as implemented in the ATLAS [81] library.

Performance can, thus, considerably change with r and c and this obviously means that making the right choice for the block size can substantially reduce the execution times for the matrix-vector product operation. Unfortunately when choosing the block size one has to take into account fill-in. The presence of fill-in increases the execution times because the overall number of floating point operations increases even if the floating point operations introduced by the presence of fill-in elements don't have any effect from an algebraic point of view. Thus, looking for the block size that yields the lower execution times actually means looking for the best compromise between high flop rates and low fill-in ratios. As an example, observe how different choices for r and c

contribute to execution times by determining different values of flop rates and fill-in for matrix number 29 (i.e. "venkat01") on the Itanium2 architecture.

Figure 3.19 reports flop rates (*top-left*), fill-in ratios (*top-right*) and execution times (*bottom-left*) for the matrix-vector product on the Itanium2 architecture with "venkat01" matrix. As pointed out before, the best choice for the block size, that is the block size that yields the lower execution times, is the one that gives the best compromise between high flop rates and low fill-in ratios; this means that the execution times plot can be seen (and in fact it is) as a merge between the other two. Even if the higher flop rate is achieved with 8×5 blocks, the lower execution time is achieved with 4×2 blocks thanks to the low (in this particular case null) fill-in. Analyzing figure 3.19, it is also possible to understand that the block size choice cannot be driven just by the knowledge of the sparsity structure of the matrix. In this case, in fact, the matrix has a regular intrinsic 4×4 block structure but this doesn't mean that 4×4 is the block size that gives the lower execution times.

The following section presents a method to exploit self-adaptativity of this matrix-vector product operation through the automatic selection of the most efficient block size. According to what said in the previous lines, given a matrix A, a straightforward approach for this automatic block-size selection would be:

1. for each $r \times c$ measure the matrix-vector flop rate $perf_A(r, c)$;
2. for each $r \times c$ measure the fill-in ratio $fill_A(r, c)$ for the BCSR representation;
3. choose the block size $r \times c$ that minimizes:

$$etime_A(r, c) \propto \frac{fill_A(r, c)}{perf_A(r, c)} \tag{3.4}$$

While this procedure certainly results in the exact selection of the block size that yields the lower execution times, it is very expensive; step one, in fact, is as expensive as computing $r \cdot c$ matrix-vector products (where other issues related to the accuracy of timing measures are not considered) and step two is as expensive as converting $r \cdot c$ matrices into BCSR format (the cost of each conversion is reported in figures 3.5 to 3.8).

The automatic selection of the block size can, by the way, be accomplished using estimates instead of the actual values of $fill_A(r, c)$ and $perf_A(r, c)$. The cost of computing these estimates is much lower based on the following considerations:

- the matrix-vector product flop rate is mostly dependent on the architecture. This means that this information can be generated once and for all the

matrices at installation time. Thus, at installation time, according to some consideration that will be discussed later, a particular reference matrix R is chosen, for each r and c (up to some limit) the performance of the matrix-vector product is measured and this information is stored in a file. Later, when selecting the block size for a particular matrix A which is given at run time, the following approximation can be assumed:

$$perf_A(r, c) \approx perf'_A(r, c) = perf_R(r, c)$$

where $perf'$ is an estimate that is equal to the actual performance measured in the case of the reference matrix R.

- matrices coming from real-world applications usually have some regularity. It could be, thus, possible to sample a matrix A with a certain accuracy, build out a submatrix A' from the chosen samples and assume:

$$fill_A(r, c) \approx fill'_A(r, c) = fill_{A'}(r, c)$$

that is, the fill-in of the whole A matrix is equal to the fill-in of a part of it.

Section 3.3.1 describes an approach in computing the fill-in estimate for a generic matrix A and in section 3.3.2 two methods of computing an estimate of the flop rates of the matrix-vector product operation in BCSR format; the former is the method presented in [79] while the latter, based on a different model, has been presented in [15].

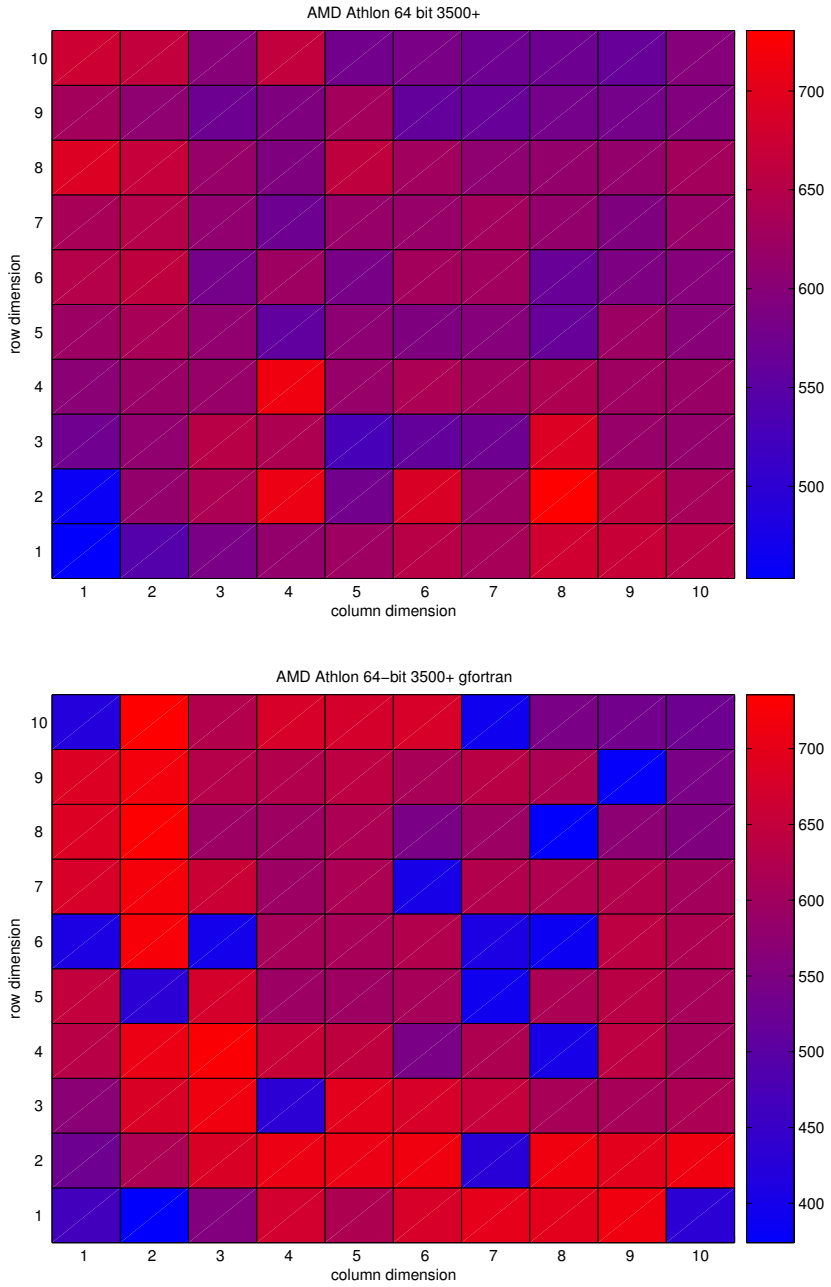

Figure 3.14: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an AMD Athlon 64-bit 3500+ architecture. The top part of the figure shows values measured on code generated by the Intel-9.0 compiler while the bottom part shows values measured on code generated by the GNU compiler v4.0.

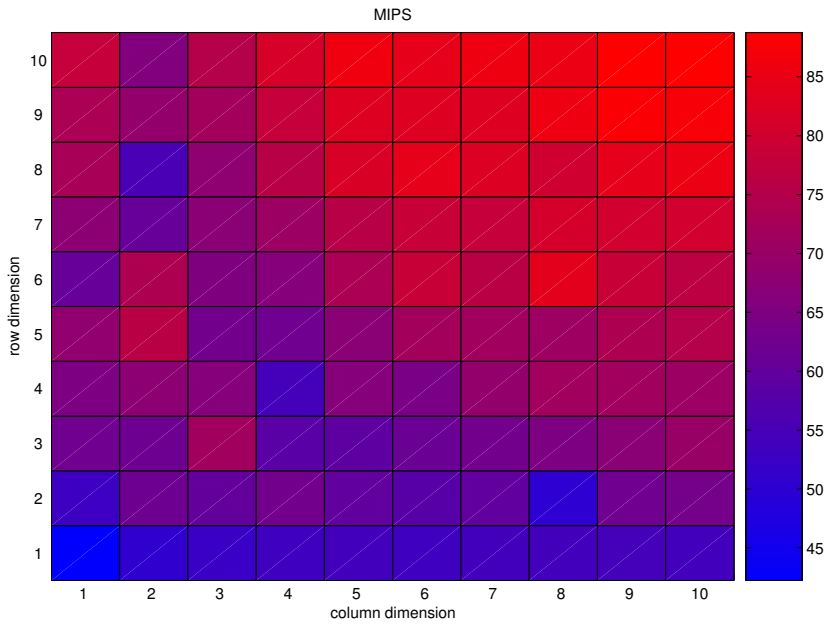

Figure 3.15: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on a MIPS architecture.

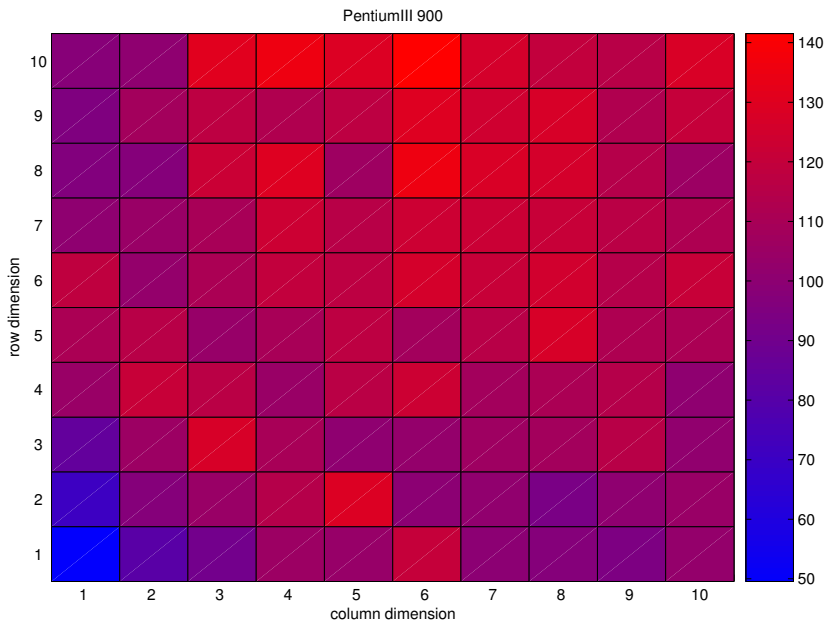

Figure 3.16: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an PentiumIII 900 architecture.

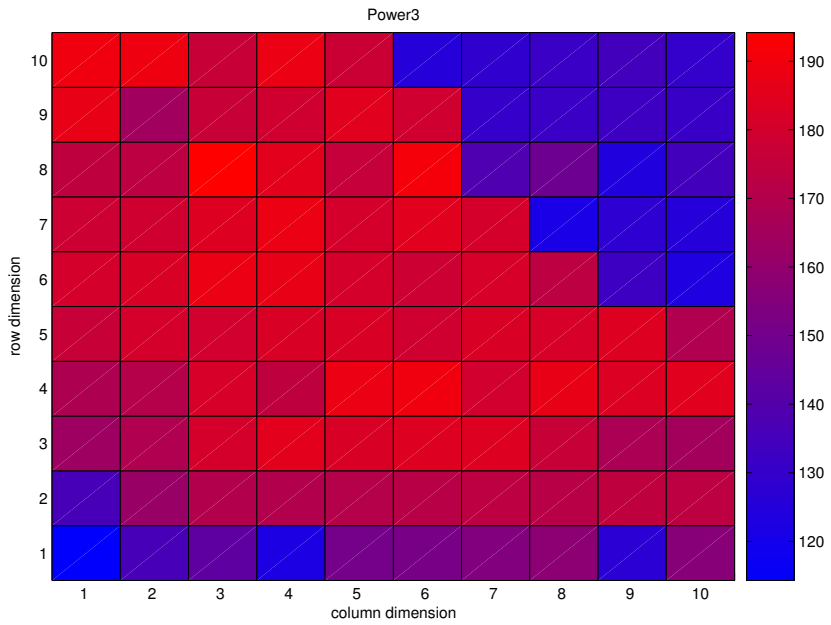

Figure 3.17: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on a Power3 architecture.

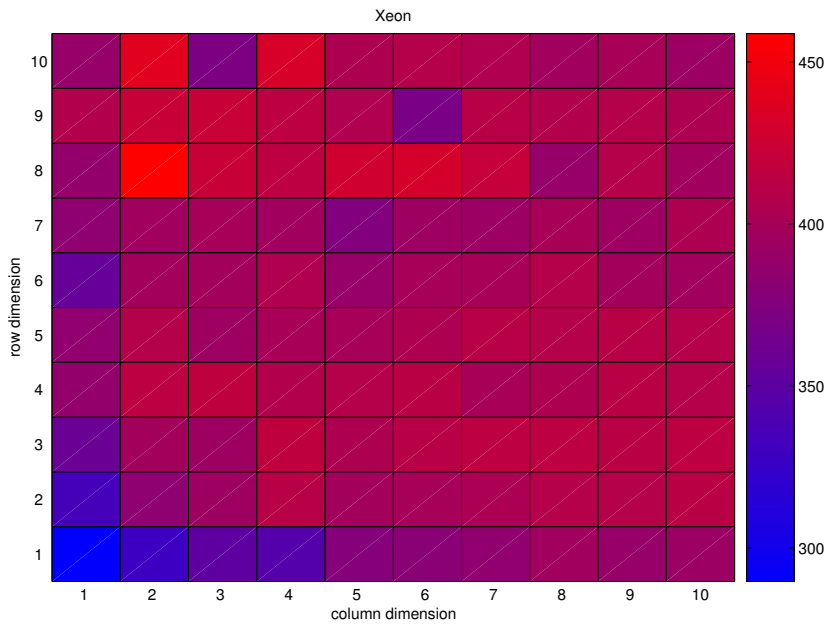

Figure 3.18: Matrix-vector product Flop rate for a 1500×1500 dense matrix stored in BCSR format on an Xeon 3060 architecture.

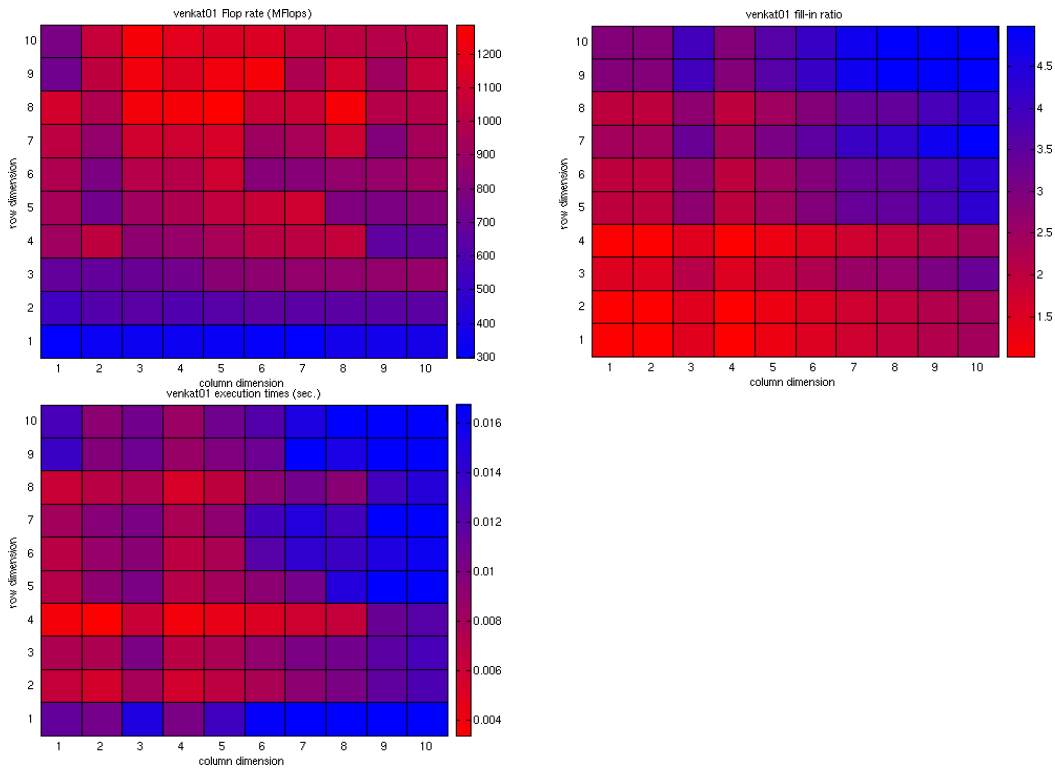

Figure 3.19: This figure shows how different choices for r and c affect execution times through flop rates and fill-in ratios. In the each part of the figure assume that "red is better" (i.e. higher flop rates, lower fill-in ratios and lower execution times). On the top-left part is depicted how flop rate changes with different values of r and c, on the top-right part is depicted how fill-in changes with r and c and on the bottom-left part is depicted how the execution times change with r and c.

3.3.1 Estimating the Fill-In

Computing an estimate of the fill-in of a given matrix is a task that must be performed at run time. This imposes a particular attention on the cost of this operation that must be reduced as much as possible to better exploit the benefits coming from the usage of the BCSR format. Most of the matrices coming from real world application has some regularity and thus it is possible to take advantage of this assumption to keep the cost of the estimate computation limited while achieving good accuracy. Based on this assumption the fill-in can be estimated sampling the matrix, computing the fill-in ratio on these samples and assume that it is the same as the fill-in ratio of the whole matrix. Algorithm 6 shows the pseudocode for the fill-in ratio estimate; *acc* is a parameter that can be fixed by the user and that defines how many samples should be taken during the estimate.

Algorithm 6 Fill-in ratio estimate pseudocode

1: compute the number of block-rows $M_{rc} = \lceil \frac{m}{r} \rceil$
2: split the set of block-rows into $n = acc \cdot M_{rc}$
3: **for** $i = 0$ to n **do**
4: randomly select a block row within the i-th set
5: compute the fill-in ratio for the selected block-row
6: **end for**
7: compute the average fill-in ratio

The cost of this operation obviously depends on the size of the matrix, on the accuracy *acc* chosen by the user and, to some extent, by the sparsity structure of the matrix. As in the operation of converting the storage format of a matrix to BCSR, also computing an estimate of the fill-in in the case of column unaligned blocks is more expensive with respect to the case where blocks are column aligned. Figures 3.20 to 3.26 show how expensive is to compute the estimate of the fill-in versus the accuracy *acc*. The cost is expressed in number of unblocked matrix-vector product. To plot these figures matrices number 28, 31 and 34 (i.e. "ct20stif", "gupta1" and "3dtube") have been choosen to compare with the data reported on [79] where the same analysis is made in the case of column aligned blocks.

The cost of this operation grows linearly with the accuracy *acc* starting from less than one unblocked matrix-vector product for very low values of accuracy up to more than a hundred unblocked matrix-vector products for 100% accuracy.

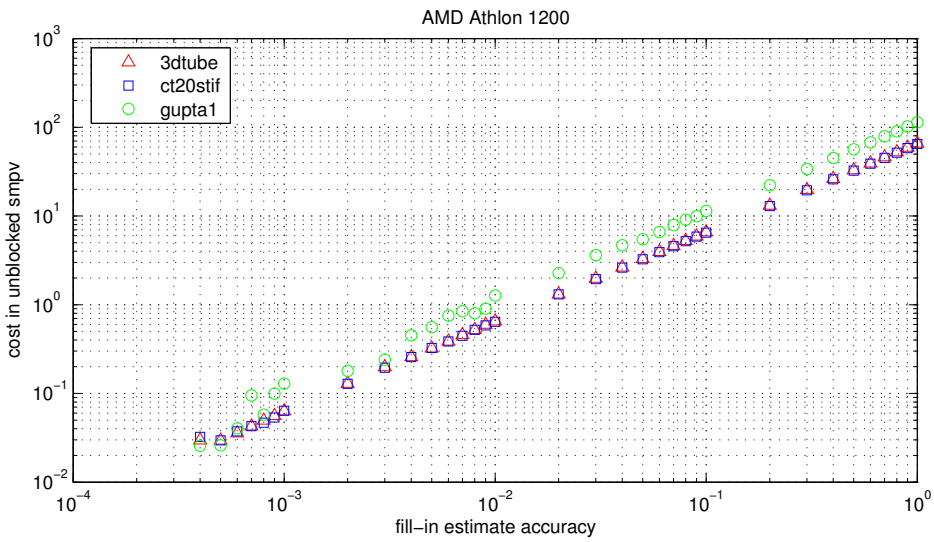

Figure 3.20: Cost of the fill-in estimate in unblocked spmv versus accuracy pf the estimate on an AMD Athlon 1200 architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

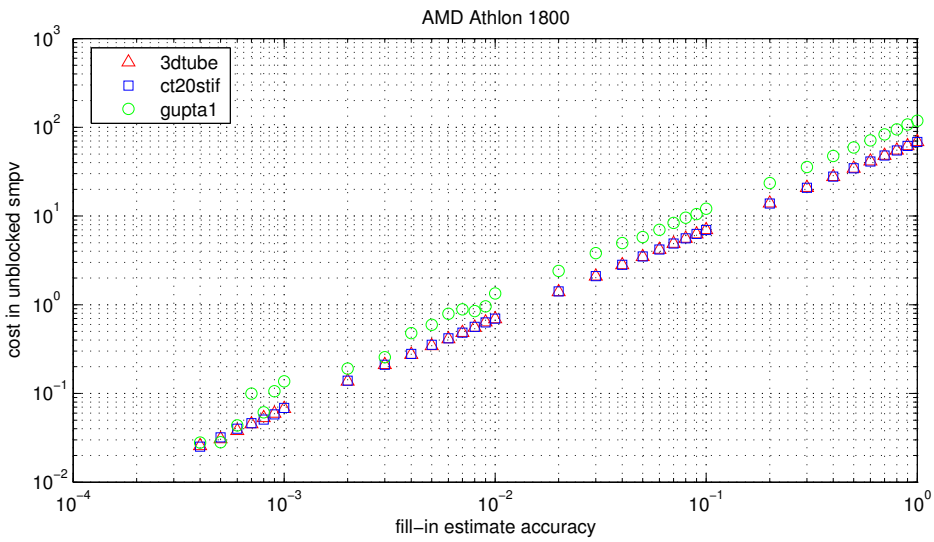

Figure 3.21: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on an AMD Athlon 1800 architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

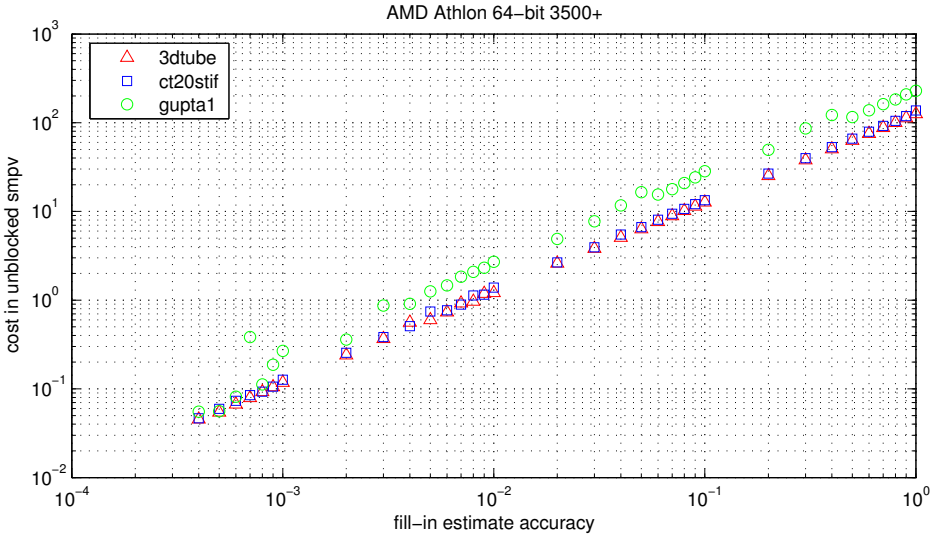

Figure 3.22: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on an AMD Athlon 64-bit 3500+ architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

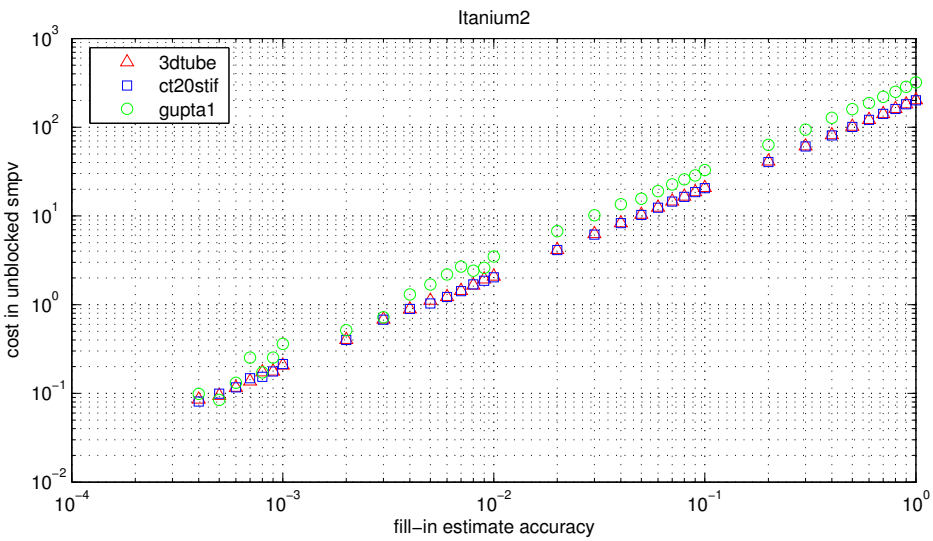

Figure 3.23: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on an Itanium2 architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

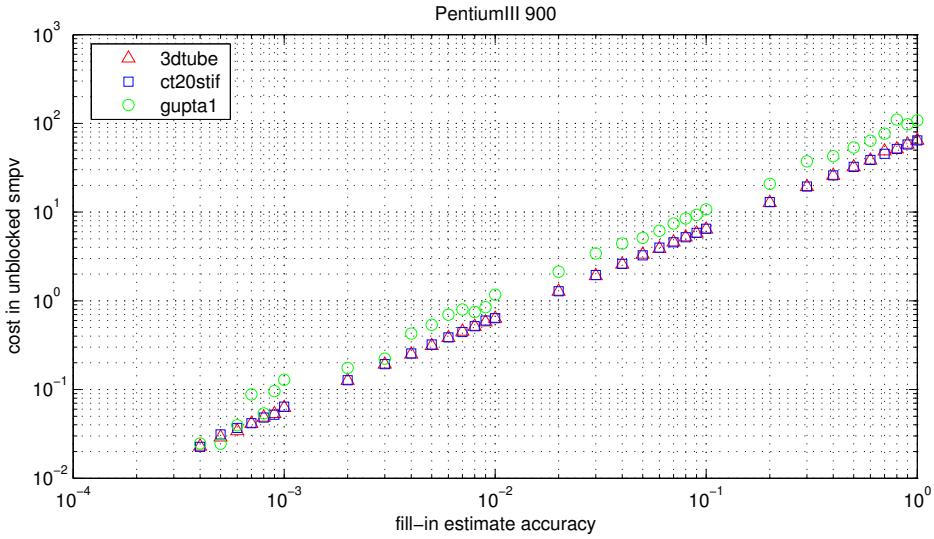

Figure 3.24: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on a PentiumIII 900 architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

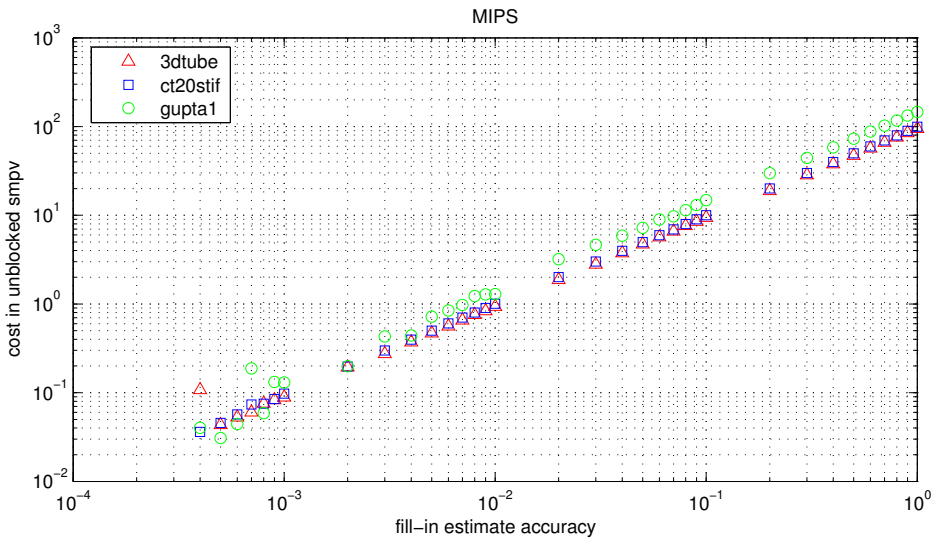

Figure 3.25: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on a MIPS architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

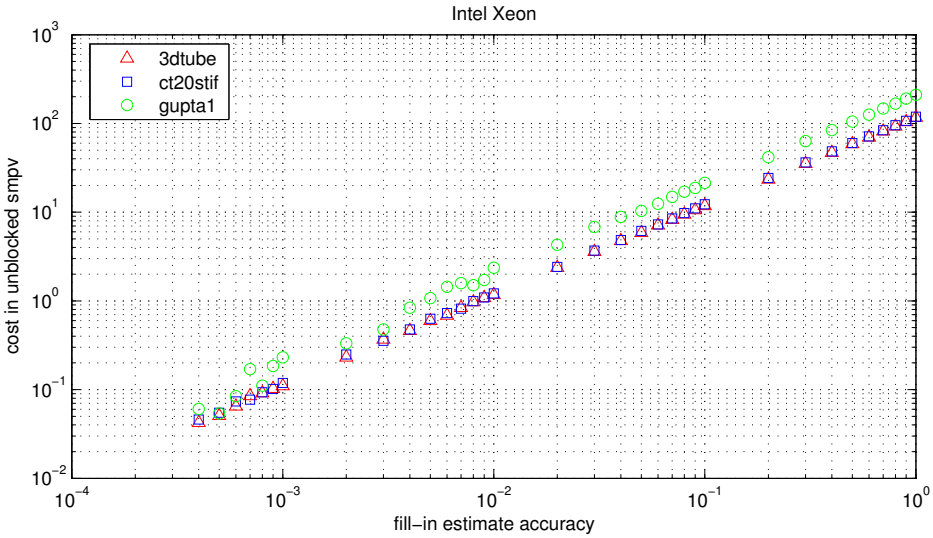

Figure 3.26: Cost of the fill-in estimate in unblocked spmv versus accuracy of the estimate on a Intel Xeon architecture. The cost is expressed in number of unblocked matrix-vector product for accuracy ranging from $4e-4$ to 1.

3.3.2 Modeling the Block Matrix Performance

The most important step in the automatic selection of the block size is building a model for the performance of the BCSR matrix-vector product. This model is used to produce an estimate of the matrix-vector product flop rate at run time when, also based on a fill-in estimate, a block size must be chosen. In the next paragraph, the model presented in [79] is briefly described and, after that, a different model is introduced. Experimental results are also reported that show how the latter performance model, presented in [15], improves the accuracy of the performance estimate.

Performance Modeling by Dense Matrix

As already discussed, the flop rates for the BCSR (with different values for r and c) matrix-vector product mostly depend on the architecture and the compiler used. Assume, for the moment, that these flop rates only depend on the architecture/compiler (the next section demonstrates how this assumption cannot be considered valid); based on this assumption, the matrix-vector product performance for a given block size $r \times c$ can be considered always the same regardless of the matrix which is being multiplied.

This suggests that the matrix-vector product performance, in general and thus also for the BCSR storage format, can be modeled by the performance of a generic reference matrix. As discussed in [79], a dense matrix is a natural choice to accomplish this task. Algorithm 7 describes the strategy used for the automatic selection of the block size as presented in [79]:

Figures 3.27 to 3.34 show the accuracy of the performance estimate computed by means of the model based on a dense matrix performance. These figures have been plotted with data measured on the 1×1 reference case; graphs related to other block size cases show exactly the same trend and, thus, are not reported here. It is possible to see that, in general, the method of performance prediction based on the dense matrix model tends to overextimate the flop rate for the matrix-vector product. There are few matrices on the PentiumIII and the MIPS architecture whose estimated performance turns out to be lower that the effective measured one. Considering that matrices in the test set have been sorted in increasing number of nonzero elements per row and looking at figures 3.27 to 3.34 it is possible to understand the performance prediction based on the dense matrix model is considerably inaccurate mostly for matrices that have few number of nonzero elements per row. Even on those architecture where the measured flop rate results to be higher than the predicted one for some matrices, the approach based on the usage of a dense matrix as a reference for the model always overextimates

Algorithm 7 Block size selection based on dense matrix model. The top part is executed once at compile time while the bottom part is the run time phase executed for each matrix.

Compile time phase

1: Build a dense reference matrix D
2: **for** $i = 1$ to max_r **do**
3: **for** $j = 1$ to max_c **do**
4: compute $perf_D(i, j)$
5: store $perf_D(i, j)$ on a file
6: **end for**
7: **end for**

Run time phase

1: Given a input matrix A
2: $mtime := \infty$
3: **for** $i = 1$ to max_r **do**
4: **for** $j = 1$ to max_c **do**
5: compute fill-in estimate $fill'_A(r, c)$
6: fetch $perf_D(i, j)$ from the file built at install time
7: assume $perf'_A(i, j) = perf_D(i, j)$
8: **if** $mtime > \frac{fill'_A(i,j)}{perf'_A(i,j)}$ **then**
9: $mtime = \frac{fill'_A(i,j)}{perf'_A(i,j)}$
10: $r = i$
11: $c = j$
12: **end if**
13: **end for**
14: **end for**
15: **return** r and c

the performance of the matrix-vector product for those matrices that are on the left part of each of the figures 3.27 to 3.34 (i.e. those with a low number of nonzero elements per row).

The following section will discuss a possible reason for this misfunction and propose a method that addresses the problem of accurately estimate the performance even for matrices that have a low number of nonzero elements per row.

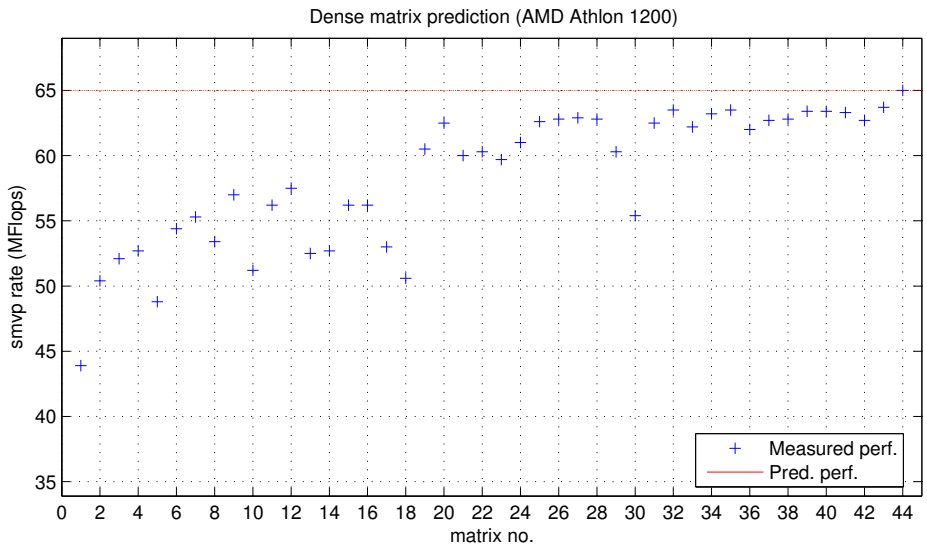

Figure 3.27: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on an AMD Athlon 1200 architecture.

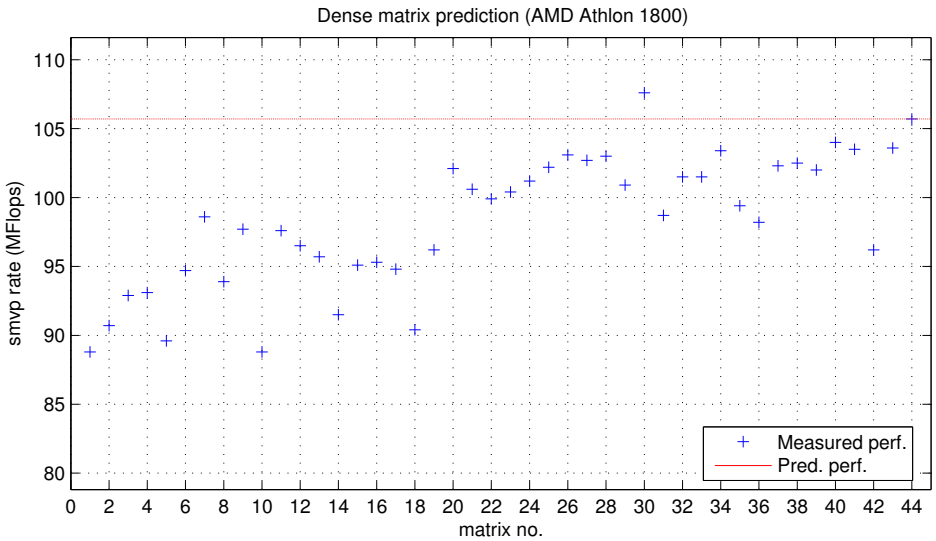

Figure 3.28: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on an AMD Athlon 1800 architecture.

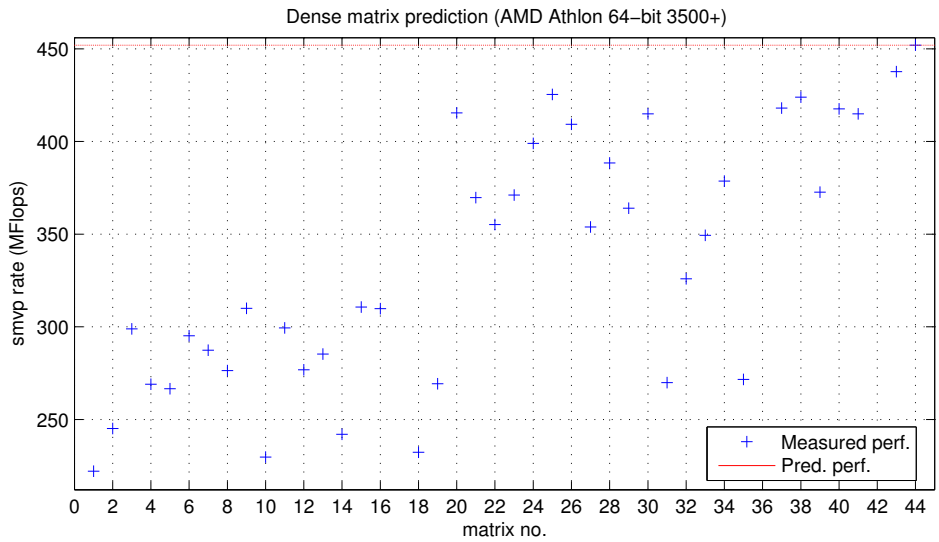

Figure 3.29: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on an AMD Athlon 64-bit 3500+ architecture.

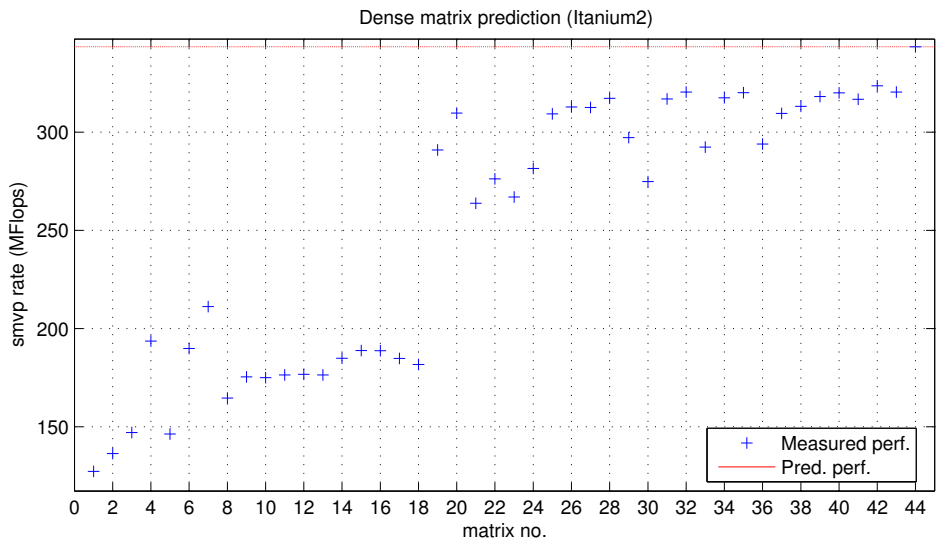

Figure 3.30: Predicted (red line)versus measured (blue "plus" signs) performance for the matrices in the testset on an Itanium2 architecture.

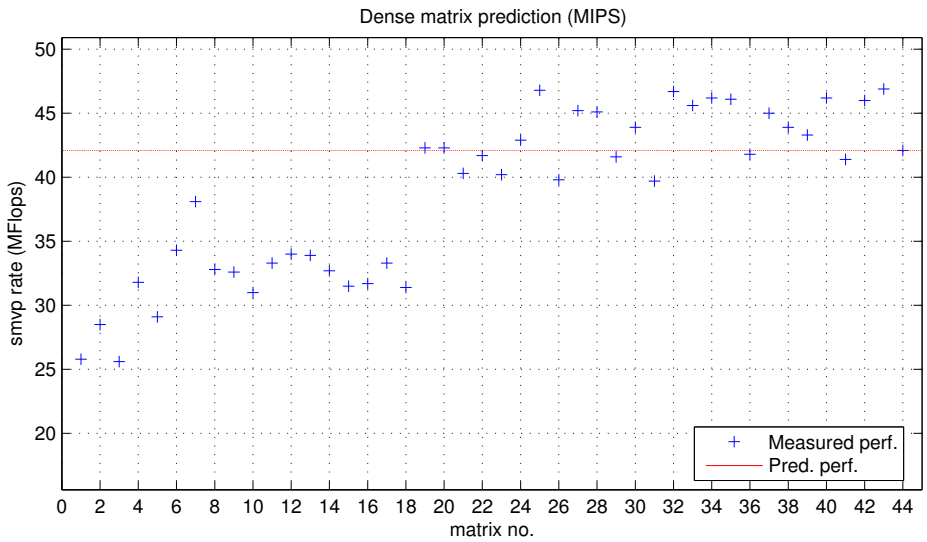

Figure 3.31: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on a MIPS architecture.

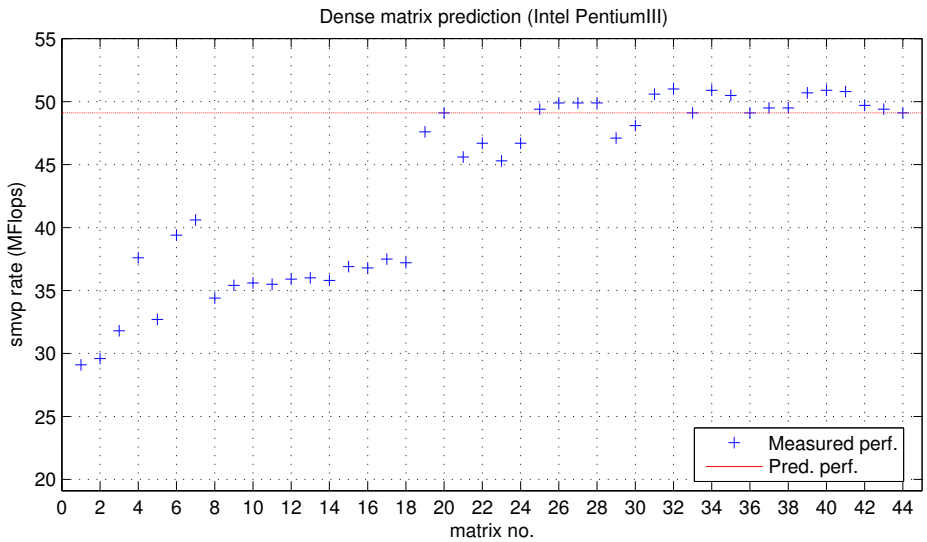

Figure 3.32: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on an Intel PentiumIII architecture.

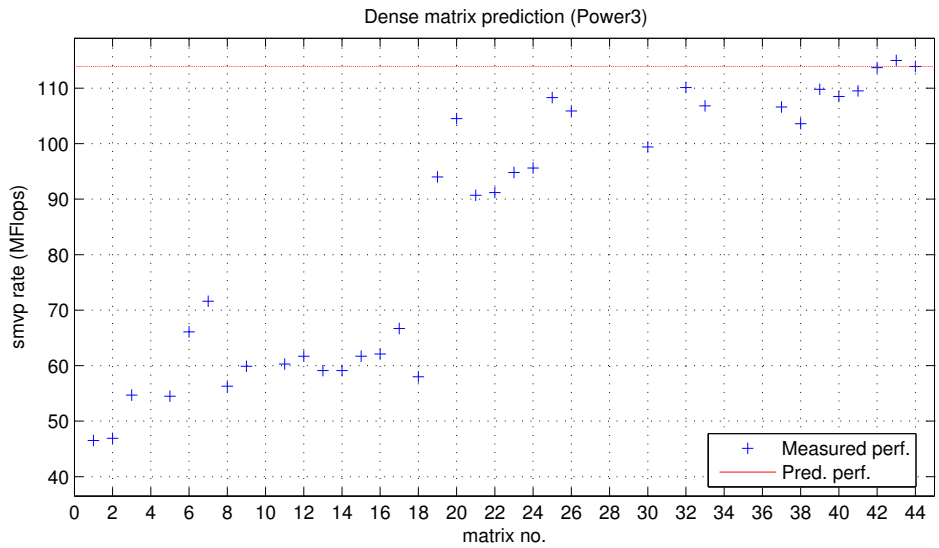

Figure 3.33: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on a Power3 architecture.

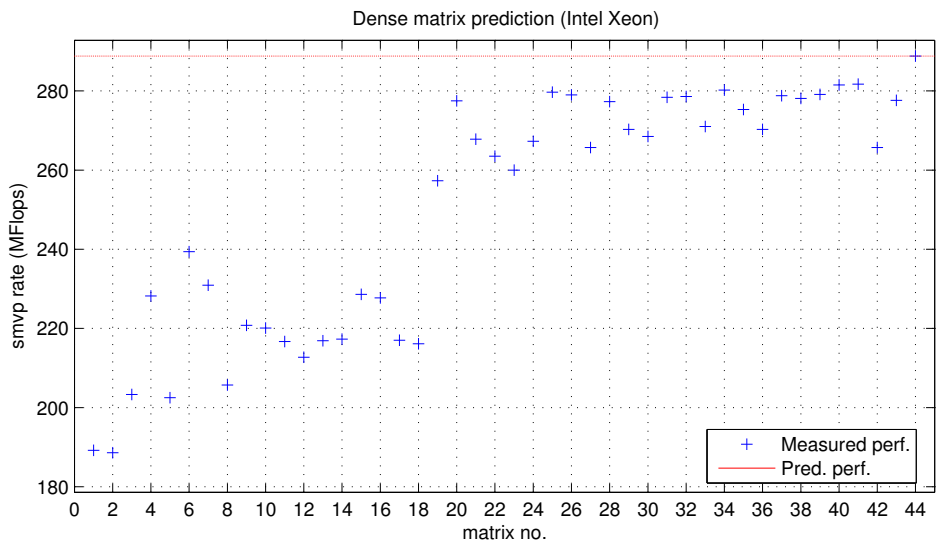

Figure 3.34: Predicted (red line) versus measured (blue "plus" signs) performance for the matrices in the testset on a Xeon architecture.

Improved Performance Model

As discussed in the previous section, the assumption that the flop rates of the matrix-vector product operation are only dependent on the machine

architecture cannot be considered valid. Analyzing figures 3.27 to 3.34 it is clear that this assumption leads to poor quality of the performance estimate in the case of matrices having a low number of nonzero elements per row (recall that the matrices the test set are sorted, and thus numbered, in ascending number of nonzero elements per row). This obviously suggests that there should be some dependency between the matrix-vector product operation performance and the number of nonzero elements per row when a compressed row storage format is used.

```
...
for(i=0;i<*m;i++,y+=1){
  register double y0=y[0];
  for(j=ia2[i];j<ia2[i+1];j++,ia1++,aspk+=1){
    y0 += aspk[0]*x[*ia1+0];
  }
  y[0]=y[0];
}
...
```

Figure 3.35: Matrix vector product source code for reference CSR implementation.

The code for the matrix-vector production the 1×1 BCSR storage format is reported in figure 3.35. The product is performed row-wise and for each row the partial result is held in an accumulator `y0`; then, at the end of the loop for a given row, the value in the accumulator is written back to memory. Thus, for each row, $2 \times nnz_{row}$ floating point operations, where nnz_{row} is the number of nonzero elements per row, and a write memory access are performed. Considering that a write memory access is much more expensive than a floating-point operation, a high ratio between the number of floating-point operations and write memory accesses obviously implies higher performance. The number of write memory accesses depends on the size of the matrix thus the matrix-vector product is likely to have better performance for those matrices which have a higher number of elements per row.

Consider, again, to the source code in figure 3.35; for each row:

- $2 * nnz_{row}$ floating point operations are performed;
- nnz_{row} integer memory reads for the elements column indices;

- $2 * nnz_{row}$ double precision memory reads, respectively one to fetch an element of the matrix and one to fetch an element of the source vector;
- one double precision memory write.

The flop rate of the matrix-vector product operation for a matrix A can be thus considered proportional to:

$$perf_A \propto \frac{nnz_{row}}{c_1 \cdot nnz_{row} + c_2} \tag{3.5}$$

where c_1 and c_2 are two constants. The green plus signs on figures 3.36 to 3.43 confirm this hypothesis. This data show the performance versus the number of nonzero elements per row for the matrices in the testset on several architectures and they are likely to follow the trend of an hyperbola like the one in equation (3.5). The data related to two block sizes (namely 1×1 and 2×3) is shown in these figures because all the other block sizes have analogous behavior.

The low flop rates on the left side of each figure show that, as expected, low number of nonzero element per row imply low performance.

The c_1 and c_2 parameters in equation (3.5) obviously depend on the architecture characteristics and the block size. The proposed approach for estimating the perfomance is based on the idea that, at istallation-time, a curve like equation (3.5) can be built for each blocks size stored on a file. This information can, then, be used at run-time (i.e. once the actual number of nonzero elements per row is known) to estimate the performance of the matrix-vector product operation. The following discussion explains how this approach is implemented in the AcCELS software package and also implicitly presents a validation of the model in equation (3.5).

Computing values for the c_1 and c_2 parameters involves measuring the performance of the matrix-vector product operation at installation time. For this purpose, the usage of matrices from real world applications (like those used to plot the green plus signs in figures 3.36 to 3.43) is not feasible mainly for two reasons:

- packaging the software becomes unpractical if large sparse matrices have to be included only to perform these installation time measures,
- even a big sample of matrices, besides being diffult and expensive to handle, can never be general enough to represent all the sparse matrices typologies.

To accomplish the task of evaluating the parameters of equation (3.5), reference matrices can be used that can be easily hand-built at installation time.

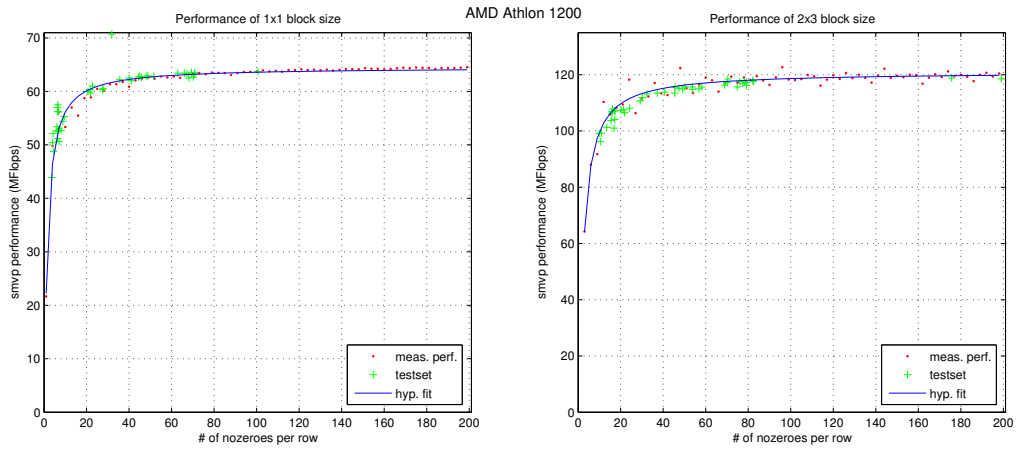

Figure 3.36: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the AMD Athlon 1200.

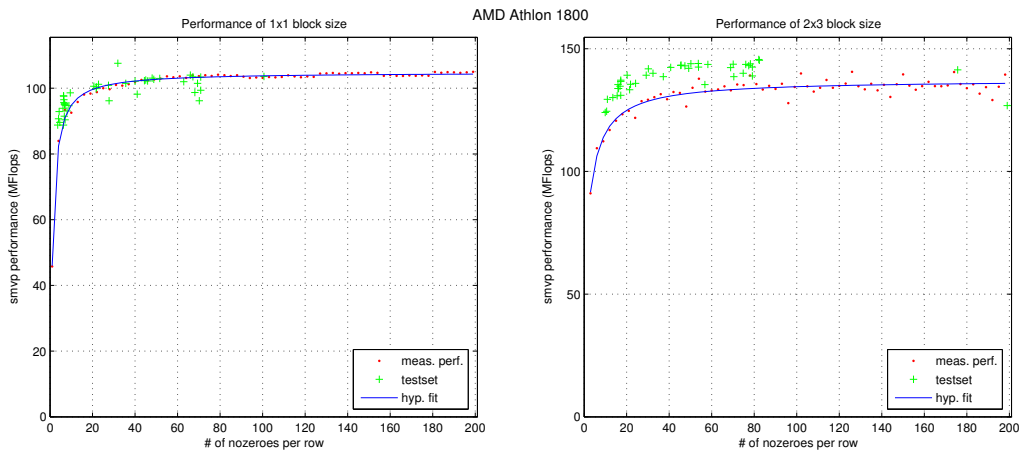

Figure 3.37: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the AMD Athlon 1800.

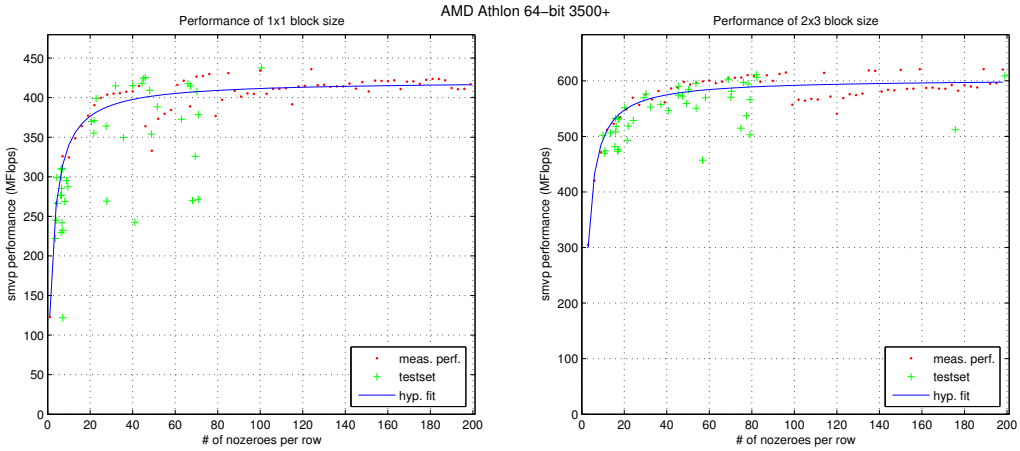

Figure 3.38: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the AMD Athlon 64-bit 3500+.

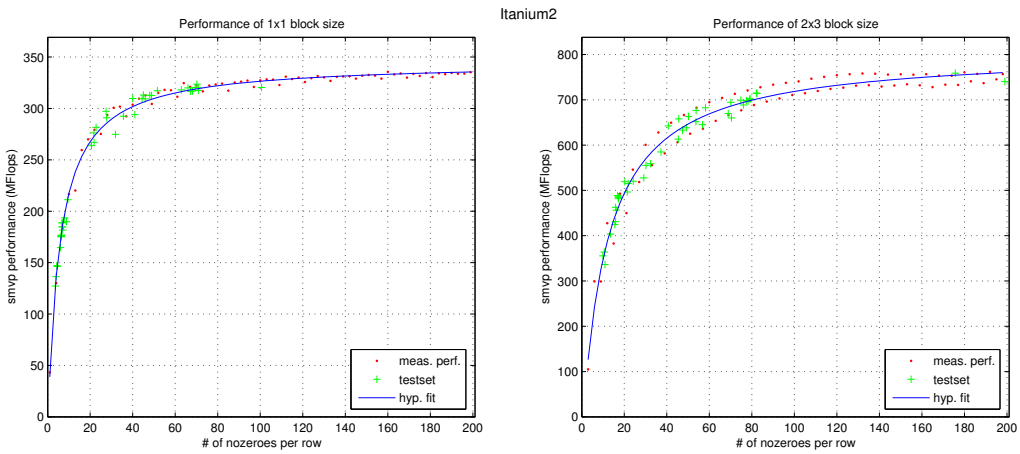

Figure 3.39: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the Itanium2.

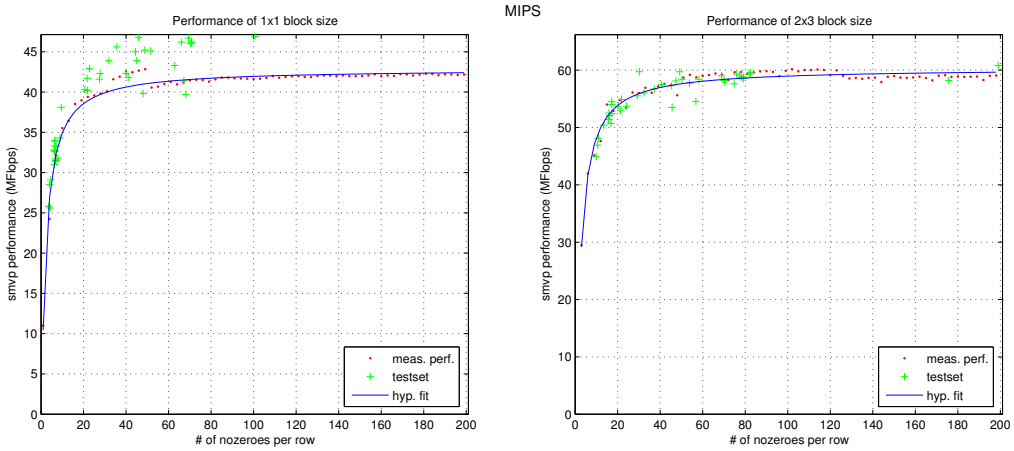

Figure 3.40: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the MIPS.

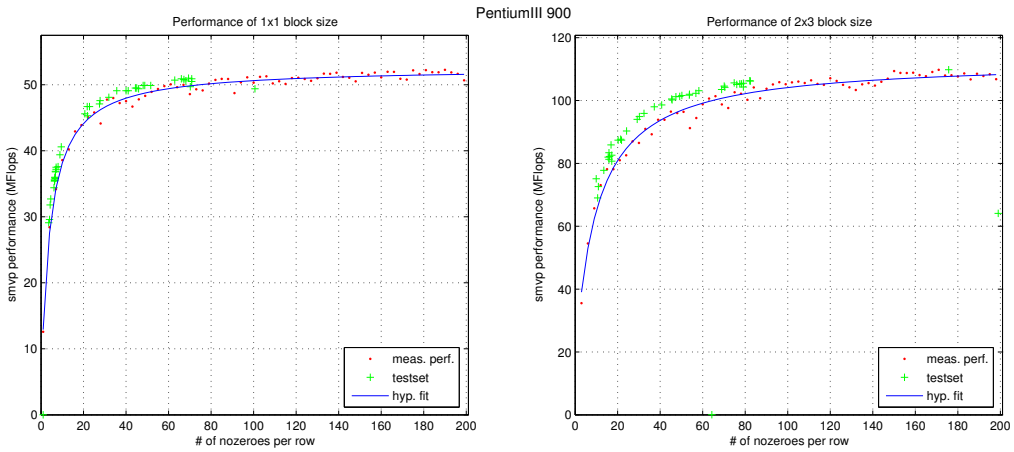

Figure 3.41: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the PentiumIII 900.

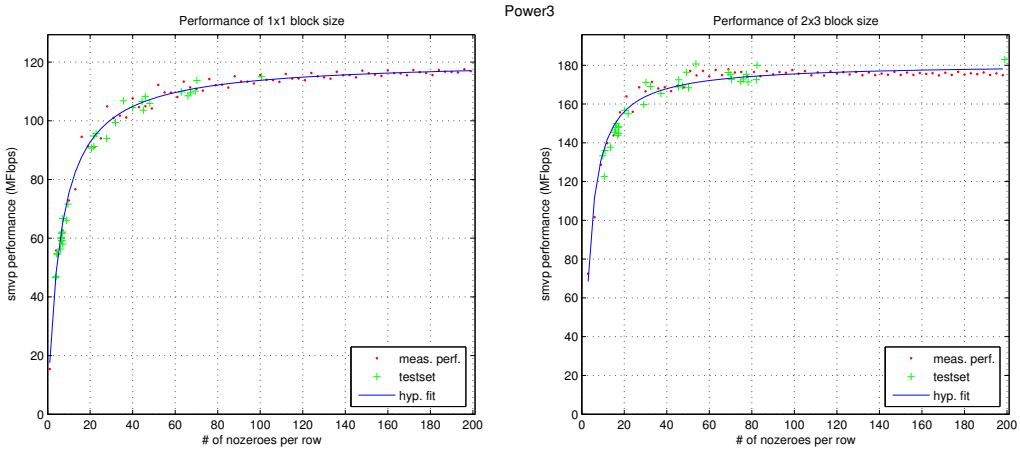

Figure 3.42: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the Power3.

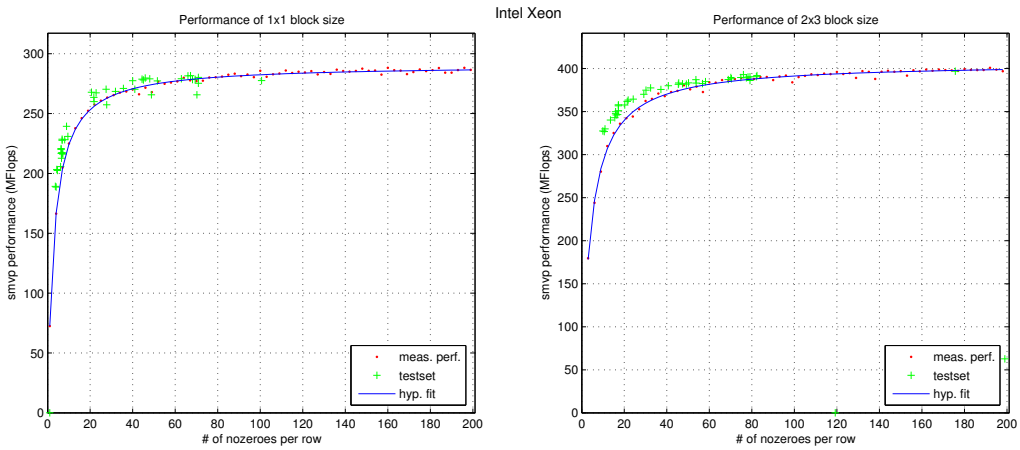

Figure 3.43: Performance versus nonzero elements per row for the 1×1 (*left*) and the 2×3 (*right*) block size cases. The green plus signs show data that is related to the matrices in out testset, the red dots show data measured on hand built banded matrices while the blue line shows how the red dots can be fitted with a hyperbola. The architecture on which this data has been measured is the Xeon.

Experimental results show that banded matrices like the one in figure 3.44 are good candidates for this purpose. These matrices present one diagonal whose bandwidth defines the number of nonzero elements per row, thus, different numbers of nonzero elements per row are obtained generating matrices with different bandwidths. Performance measures for these banded matrices are plotted by the red dots in figures 3.36 to 3.43 for $1 \leq nnz_{row} \leq 200$.

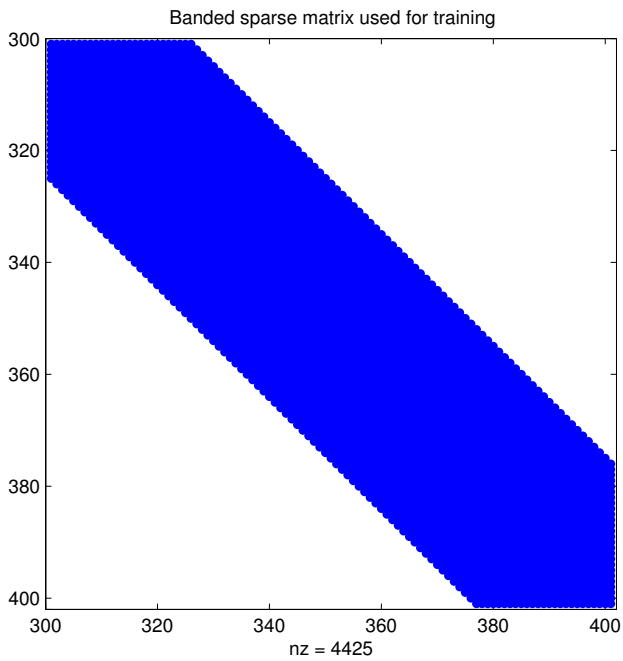

Figure 3.44: A portion of a banded matrix used during the training stage. These matrices have one diagonal whose bandwidth is varied to measure how flop rates change with a different number of nonzero elements per row.

It must be noted that banded matrices represent an optimistic case because of their excellent properties in terms of cache behavior: elements along each row are in contiguous positions providing high spatial locality on the source vector elements and the number of elements with the same column index is high providing high temporal locality for the elements of the source vector. Figures 3.36 to 3.43 show, anyway, that these banded matrices behave almost the same as matrices coming from real world applications like those in the testset. Moreover exploring the way elements are distributed along each row adds more complexity to the run time phase which is, obviously, undesirable because the automatic block size selection should be performed as fast as

possible.

A validation of the models can be accomplished by fitting the red dots in figures 3.36 to 3.43 with a function of the type (3.5). Anyway the function used in the AcCELS package for this purpose is of the type:

$$perf_A \propto a + \frac{b}{nnz_{row} + c} \tag{3.6}$$

because it better fits the data and because experimental tests show that $ac = -b$ within few percents. The blue curves in figures 3.36 to 3.43 show how the red dots can be fitted with an hyperbola function like the one in 3.6. These curves show that the performance of the matrix-vector product, which is a function of the number of nonzero elements per row, can be modeled with an hyperbola like (3.6) (or (3.5) according to the considerations above).

The fitting procedure is also used at istallation time to build the hyperbola associated to each block size but, for these purpose, only few matrix-vector products have to be performed (5 or 6).

Algorithm 8 shows the pseudo-code describing the actions executed at both the installation (*top*)and run-time phase (*bottom*) to perform the automatic block size selection.

Comparing the method described in algorithm 8 to the one in 7, the following considerations can be done:

- the accuracy of the performance prediction is improved (section 3.3.3 will give experimental verification of this assertion) by the fact that the new model is aware of the number of nonzero elemtns per row,

- the cost of the installation time is higher with respect to the case of the dense matrix based model. In the latter only one matrix is generated at installation time (i.e. the dense reference matrix) and only one matrix-vector operation is performed for each block size. In the case of the hyperbola based performance model, for each block size, a banded matrix has to be generated for at least 5 or 6 different numbers of nonzero elements per row (i.e. a number of points that is enough to accurately perform the fitting procedure) and the matrix-vector product operation has to be run on all of them to identify few points of the associated hyperbola; the fitting has, then, to be performed on these points to compute the parameters a, b and c for the block size with which the points have been measured. Anyway this increased cost is negligible since the installation is performed just once.

- the cost of the run time phase is almost the same as in the dense matrix based prediction. Once the a, b and c parameters have been fetched

Algorithm 8 Block size selection based on hyperbola model. The top part is executed once at compile time while the bottom part is the run time phase executed for each matrix.

Compile time phase

1: **for** $i = 1$ to max_r **do**
2: **for** $j = 1$ to max_c **do**
3: **for** $nzr = j$ to 200 step $j \cdot \lfloor 20/j \rfloor$ **do**
4: build a matrix R with nzr nonzeroes per row
5: compute $perf_R(i, j, nzr)$
6: **end for**
7: fit the points with an hyperbola $a + \frac{b}{x+c}$
8: store a_{ij},b_{ij} and c_{ij} on file
9: **end for**
10: **end for**

Run time phase

1: Given a input matrix A with nzr nonzeroes per row
2: $mtime := \infty$
3: **for** $i = 1$ to max_r **do**
4: **for** $j = 1$ to max_c **do**
5: compute fill-in estimate $fill'_A(r, c)$
6: $nzr' = nzr \cdot fill'_A(r, c)$ {the nnz_{row} increases due to fill-in}
7: fetch a_{ij}, b_{ij} and c_{ij} from the file built at install time
8: compute $perf'_A(i, j) = a_{ij} + \frac{b_{ij}}{nzr'+c_{ij}}$
9: **if** $mtime > \frac{fill'_A(i,j)}{perf'_A(i,j)}$ **then**
10: $mtime = \frac{fill'_A(i,j)}{perf'_A(i,j)}$
11: $r = i$
12: $c = j$
13: **end if**
14: **end for**
15: **end for**
16: **return** r and c

from file for a block size the performance value is estimated through the formula $a + b/(nnz_{row} + c)$. The added cost for the run time phase is just three floating point operations which is absolutely negligible if compared to the cost of the fill-in estimate.

Algorithm 8 also show that for real sparse matrices even higher speedups

can be obtained on the matrix-vector product operation flop rate than in the dense case due to the fill-in. Assume, for example, that matrix A has nzr nonzero elements per row when represented in CSR format. When representing matrix A in BCSR format the fill-in $fill_A(r, c)$ is likely to be greater than one; this means that the number of nonzero elements per row increases by a factor equal to the fill-in ratio. The new number of nonzero elements per row actually becomes $nzr' = fill_A(r, c) \cdot nzr$.

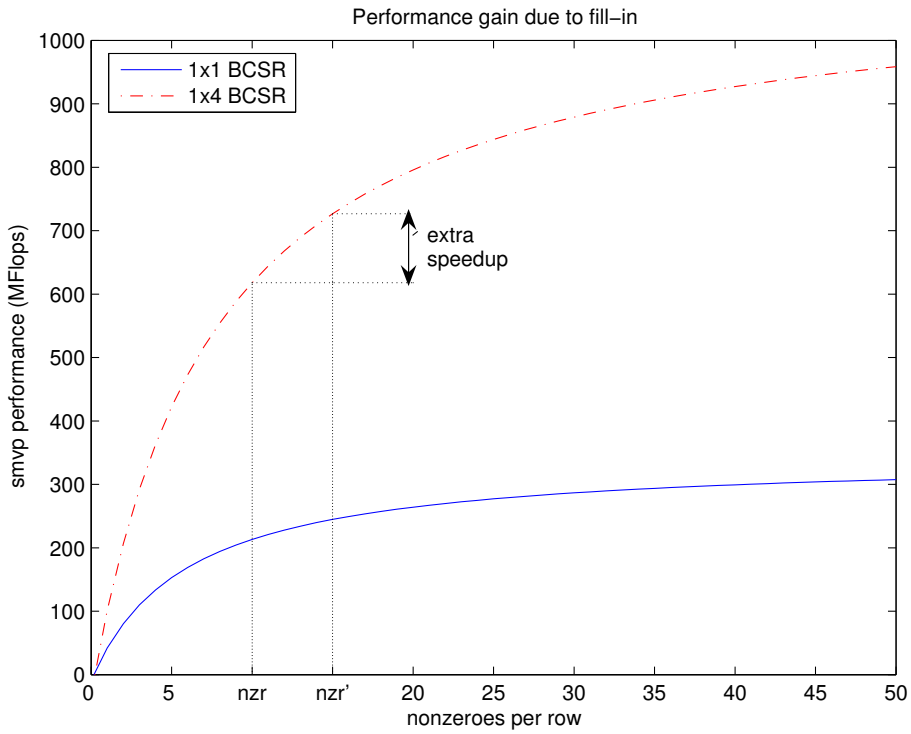

Figure 3.45: Having a fill-in ratio greater than one causes an increase in the matrix-vector product performance due to the fact that number of elements per row has grown. Thus, moving from the CSR representation to the BCSR representation with $r \times c$ blocks, there is an overall speedup that is defined by the sum of the speedup coming from register reuse, loop unrolling etc. and the extra speedup shown in this figure. Here $nzr' = fill_A(r, c) \cdot nzr$.

Figure 3.45 shows that the speedup of the BCSR format with respect to the CSR one is the sum of the speedup coming from register reuse, loop unrolling etc. (see section 3.3) and an extra speedup due to an increase in the number of nonzero elements per row.

Architecture	Dense	Hyperbola
AMD 1200	0.0906	0.0233
AMD 1800	0.0447	0.0224
AMD 64-bit	0.0844	0.0369
Itanium2	0.2326	0.0212
MIPS	0.0771	0.0313
PentiumIII	0.2963	0.2148
Power3	0.0818	0.0506
Xeon	0.0626	0.0225

Table 3.2: This table shows the average error on the performance prediction with the dense matrix based approach and the hyperbola based approach on the machines used.

The proposed approach has been tested on the matrices described in appendix A. The average error on performance estimation for the matrices is reported in table 3.2. Performance estimate accuracy is considerably improved with respect to the method based on the dense matrix model: in the case of the Itanium2 architecture, the error is reduced by a factor of ten, from 23% to 2%; the worst case is on the PentiumIII 900 architecture where still the error is reduced from 29% to 21%.

Figures 3.46 to 3.53 show the accuracy of the two discussed performance prediction methods on the machines and the matrices described in appendix A. Recall that matrices are numbered in ascending order of nonzero elements per row. It is possible to see that, for the dense matrix based model, the error on the performance estimate can be fitted with a hyperbola function of the type (3.6) plotted by the red line. Fitting the results obtained with the hyperbola based model, instead, yields a straight line (except in the case of the PentiumIII 900 machine) meaning that this approach obtains the same accuracy regardless the number of nonzero elements per row. Figure 3.49 shows that the hyperbola based approach yields considerably good results on the Itanium2 architecture where, in fact, as it can be seen in figure 3.39, the distance between the beginning and the asymptote of the hyperbola is higher with respect to the other architectures.

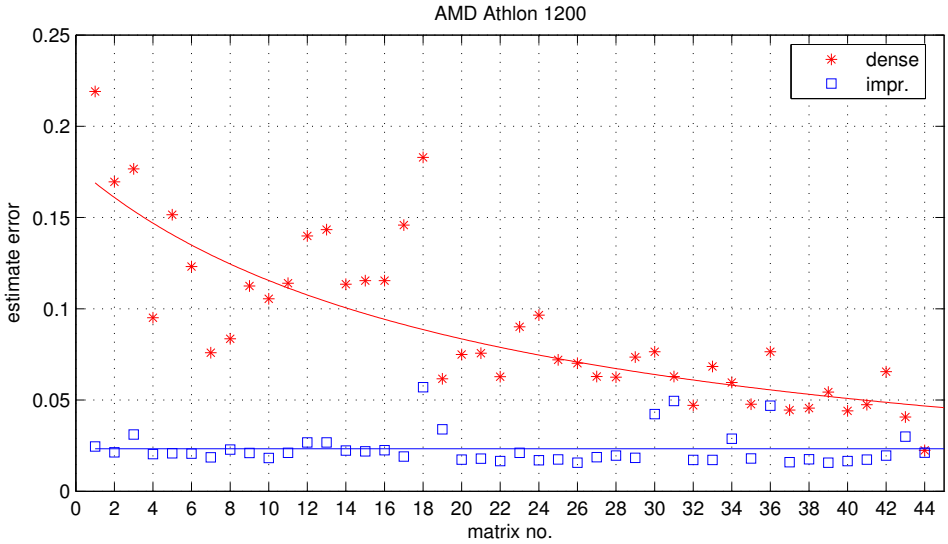

Figure 3.46: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on an AMD Athlon 1200 machine. Points are fitted respectively with the *red* and the *blue* curves.

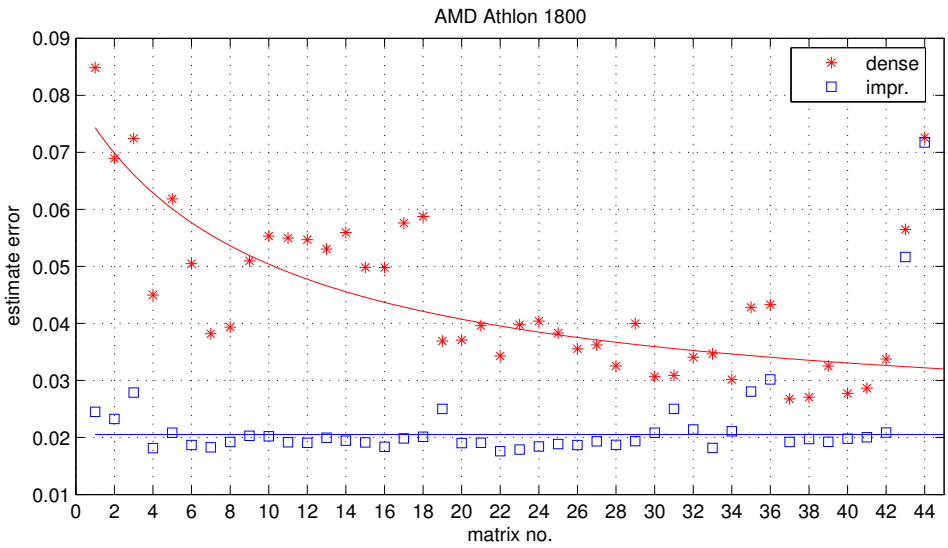

Figure 3.47: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on an AMD Athlon 1800 machine. Points are fitted respectively with the *red* and the *blue* curves.

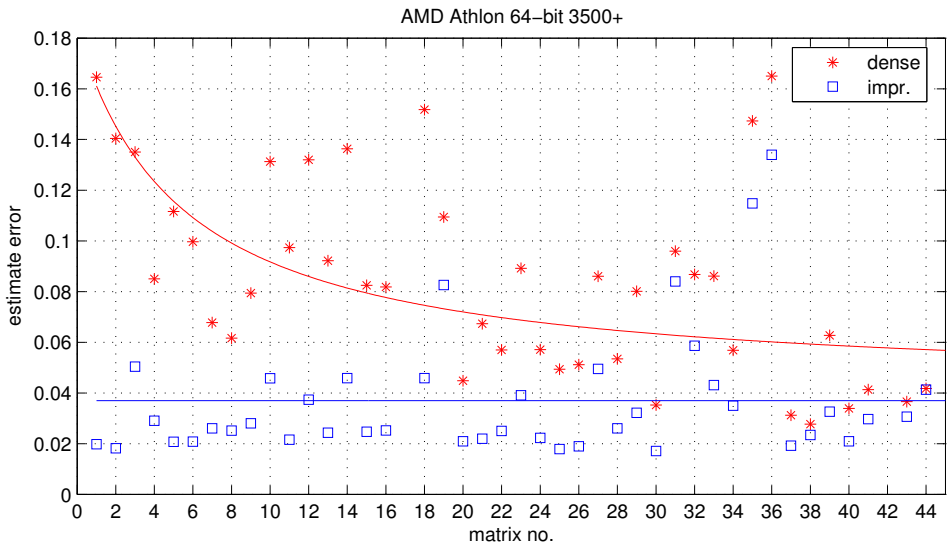

Figure 3.48: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on an AMD Athlon 64-bit 3500+ machine. Points are fitted respectively with the *red* and the *blue* curves.

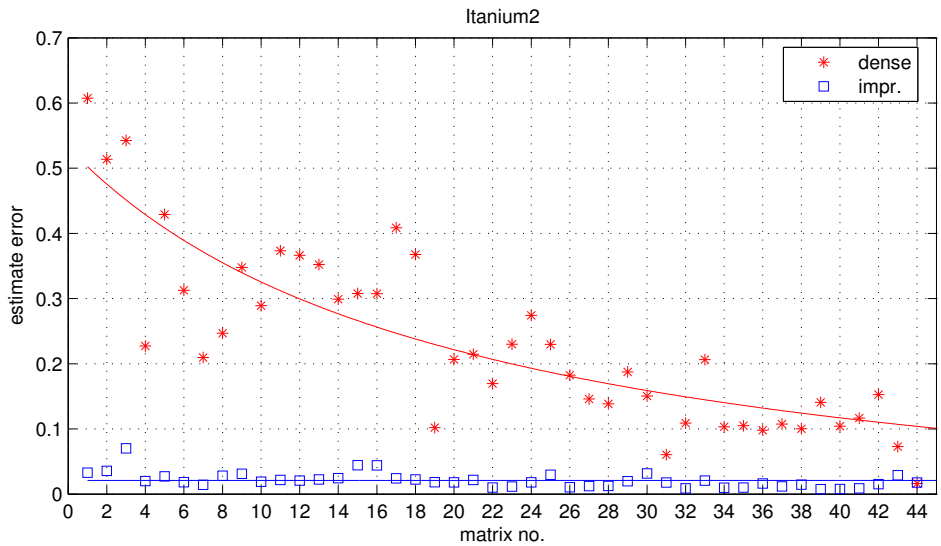

Figure 3.49: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on an Itanium2 machine. Points are fitted respectively with the *red* and the *blue* curves.

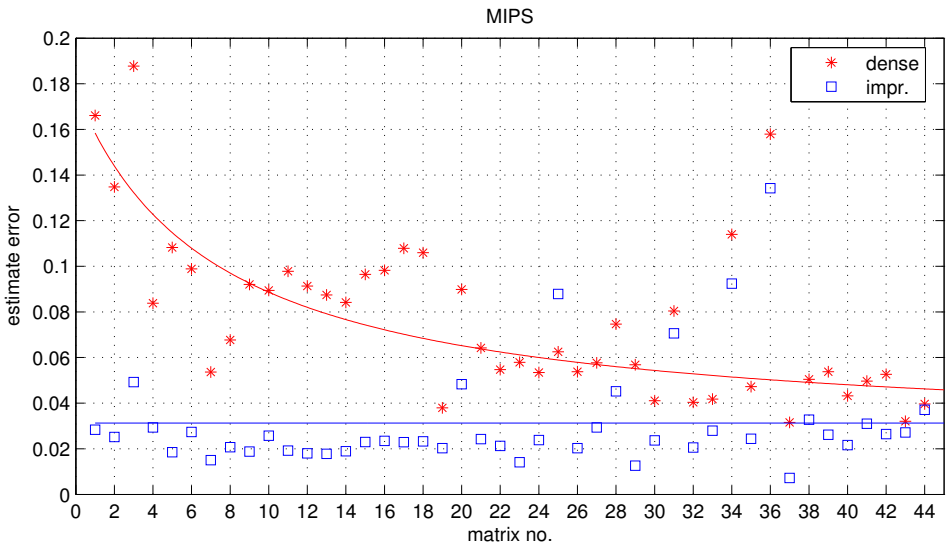

Figure 3.50: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on a MIPS machine. Points are fitted respectively with the *red* and the *blue* curves.

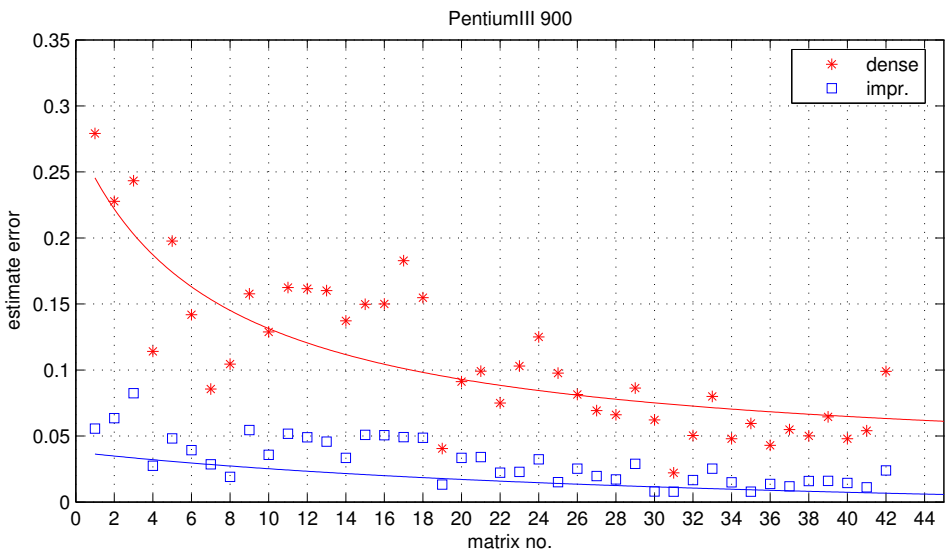

Figure 3.51: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on a PentiumIII 900 machine. Points are fitted respectively with the *red* and the *blue* curves.

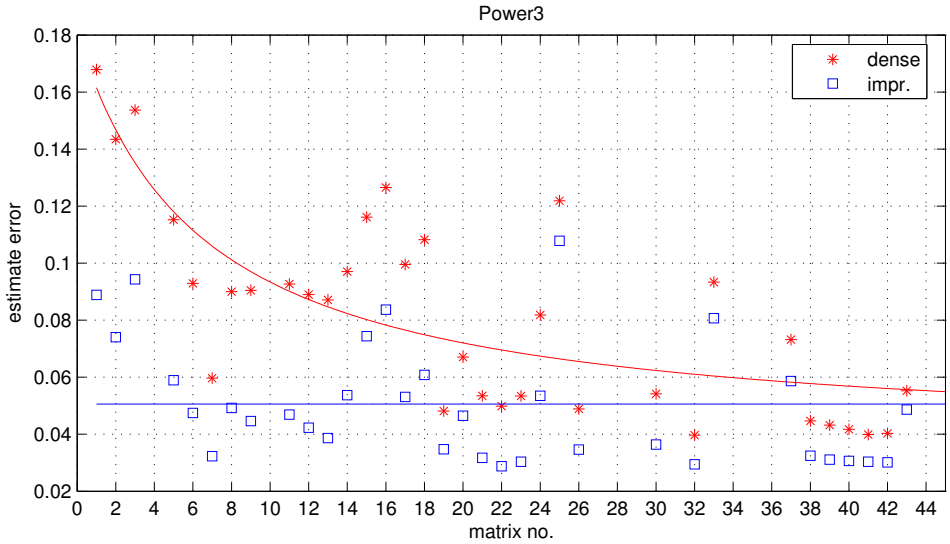

Figure 3.52: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on a Power3 machine. Points are fitted respectively with the *red* and the *blue* curves.

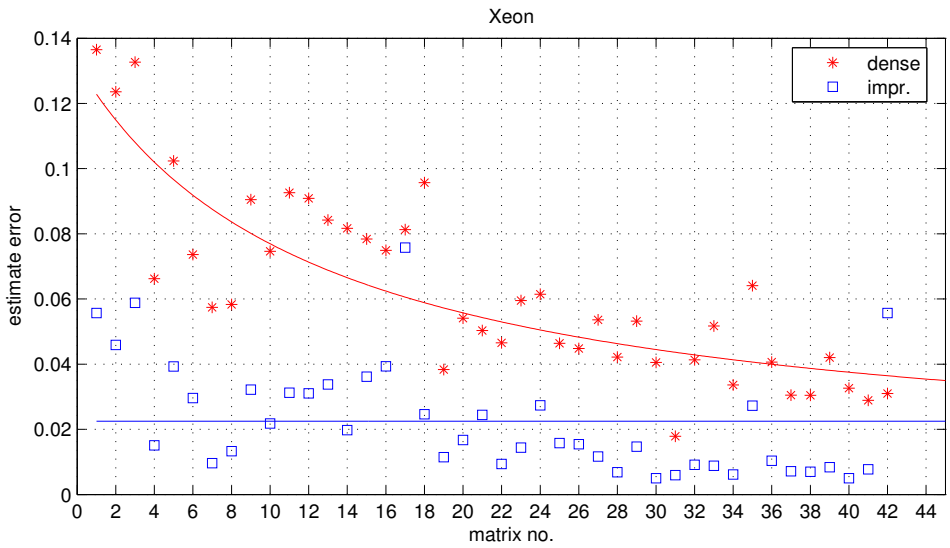

Figure 3.53: Estimate error with the dense matrix based model (*red stars*) and with the hyperbola based model (*blue squares*) on an Intel Xeon machine. Points are fitted respectively with the *red* and the *blue* curves.

3.3.3 Results

The AcCELS package has been tested on the matrices and the architecture described in appendix A. The timings measured on this test runs are reported in this section.

Table 3.3 shows the speedup that can be achieved on the matrix-vector product operation using the BCSR storage format as implemented in the AcCELS package with respect to the reference CSR storage format. Again, the architecture that gives the best results is the Intel Itanium2 where the speedup can be as big as 3.45 with the "venkat01" matrix, where the dense matrix has been excluded for obvious reasons. The Intel Xeon architecture is the one that seems achieve less improvements by the usage of the hyperbola based performance prediction method. In some cases the usage of the BCSR storage format results to be not convenient; these are the cases in table 3.3 where the speedup is lower than 1. The occurrence of these cases reflect the fact that either fill-in or performance estimate may be inaccurate resulting in a wrong block size selection. This loss of efficiency is, however, just sporadic and within few percents of the reference CSR performance.

It is important to note that using the hyperbola based model in performance estimate comes at the cost of three floating-point operations when compared to the dense matrix based one (i.e. the cost of computing the performance rate with the formula in (3.6)). The cost of building and fitting the hyperbola can be considered not relevant because these operations are performed at installation time.

Figures 3.54 to 3.61 show the time spent in the matrix-vector product with the CSR reference implementation (*black cross*) and with the BCSR storage format both in the case where the block size has been selected using the dense matrix based performance prediction (*red stars*) and the hyperbola based performance prediction (*blue squares*) models. Values have been normalized to the time measure for the best case BCSR block size. The following information can be extracted by these pictures:

- improvement with respect to the dense based performance prediction model: blue squares are systematically above the red stars except few exceptions. This means that, even when mitigated with the inaccuracy of the fill-in estimate model, the advantages coming the the usage the hyperbola based performance prediction model are still considerable. These advantages are more evident on the Itanium2 and Power3 architecture while on other machines the two approaches behave almost the same.

- the automatic block size selection often results in the best block size case or very close to it. This means that the performance prediction

Mat. no.	AMD 1200	AMD 1800	AMD 64-bit	Itanium2	MIPS	PentiumIII	Power3	Xeon
1	1.00	1.00	1.00	1.74	1.00	0.97	1.16	1.00
2	0.93	1.00	1.00	1.76	1.00	1.08	1.26	1.00
3	0.84	1.00	1.00	1.93	0.88	1.12	1.06	1.00
4	1.00	1.00	1.00	1.48	1.00	1.00	=	1.00
5	1.00	1.00	1.00	1.69	1.00	0.95	1.09	1.00
6	1.09	1.00	1.00	1.90	1.00	1.07	1.23	1.00
7	1.00	1.00	1.00	1.15	1.00	1.00	1.00	1.00
8	1.00	1.00	1.00	1.22	1.00	1.00	1.03	1.00
9	1.00	1.00	1.00	1.66	1.00	1.09	1.24	1.00
10	1.00	1.00	1.00	1.65	1.00	1.00	=	1.00
11	1.04	1.00	1.00	1.63	1.00	1.12	1.25	1.00
12	0.91	1.00	1.00	1.59	1.00	1.05	1.08	1.00
13	1.00	1.00	1.00	1.52	1.00	1.00	1.03	1.00
14	1.28	1.11	1.13	2.13	1.19	1.42	1.64	1.19
15	1.01	1.00	1.00	1.66	1.00	1.05	1.23	1.00
16	1.03	1.00	1.00	1.66	1.00	1.05	1.19	1.00
17	1.07	1.00	1.00	1.90	1.00	1.09	1.22	1.00
18	1.08	1.00	1.00	1.75	0.98	1.19	1.40	1.00
19	1.00	1.00	1.00	1.23	1.00	1.00	1.00	1.00
20	2.46	1.34	1.45	2.77	1.56	2.28	1.86	1.44
21	1.56	1.07	1.11	2.36	1.13	1.49	1.33	1.09
22	1.12	1.00	1.00	1.58	1.00	1.15	1.00	1.00
23	1.31	1.02	1.07	1.80	1.00	1.27	0.99	1.03
24	1.56	1.12	1.13	2.41	1.21	1.65	1.45	1.12
25	2.39	1.30	1.33	2.68	1.48	2.00	1.68	1.25
26	2.80	1.50	1.41	2.97	1.62	2.21	1.70	1.40
27	2.75	1.47	1.48	2.68	1.39	2.18	=	1.39
28	1.89	1.31	1.28	2.61	1.16	1.84	=	1.22
29	2.38	1.45	1.66	3.48	1.60	2.21	=	1.44
30	0.96	1.00	1.00	1.59	1.00	1.11	1.00	1.00
31	0.97	1.00	1.00	1.36	1.00	1.06	=	1.00
32	2.71	1.45	1.73	2.66	1.46	2.34	1.76	1.46
33	1.63	1.09	1.18	2.04	0.97	1.51	1.22	1.07
34	2.13	1.28	1.52	2.50	1.45	2.31	=	1.41
35	2.72	1.45	2.03	2.68	1.47	2.34	=	1.46
36	1.58	1.11	1.36	1.90	1.16	1.52	=	1.20
37	1.35	1.00	1.00	1.86	1.00	1.21	1.04	1.00
38	1.36	1.00	1.00	1.87	1.00	1.21	1.03	1.00
39	2.81	1.50	1.45	2.80	1.49	2.20	1.60	1.33
40	1.91	1.31	1.26	2.54	1.14	1.82	1.33	1.22
41	1.84	1.36	1.28	2.59	1.32	1.91	1.40	1.26
42	3.64	1.80	1.63	4.09	1.74	2.46	1.69	1.59
43	1.40	1.04	1.00	1.09	1.00	1.31	1.00	1.00

Table 3.3: This table reports the speedup data for the BCSR matrix-vector with respect to the reference CSR case. The BCSR block size has been selected with the hyperbola based model for the performance prediction.

model and the fill-in estimate are accurate. On the Itanium2 and Power3 architectures the dense based performance prediction model results to be considerably less efficient than the hyperbola based one leading to the selection of block sizes that yield performance results substantially far from the best block size case.

- using the BCSR storage format is considerably more convenient with matrices that have a high number of nonzero elements per row. Recall that matrices in the testset are sorted in ascending order of nonzero elements per row. Figures 3.54 to 3.61 show that the distance between the CSR timings (*black cross*) and the BCSR ones (either *red stars* or *blue squares*) is higher in the right side of each picture. This is due to the fact that, when the number of nonzero elements per row is very low, the fill-in grows very rapidly with the block size preventing the possibility to use block sizes that can be very fast.

The speedup shown in table 3.3 as well as in figures 3.54 to 3.61 has to be analyzed in conjunction with the data plotted in figures 3.20 to 3.26. The cost of estimating the fill-in, in fact, makes the BCSR storage format only convenient in those applications where a high number of matrix-vector products has to be performed. For example, the speedup obtained on matrix number 28 (i.e. "ct20stif" matrix) on the Itanium2 architecture is 2.61; estimating the fill-in for this matrix on the Itanium2 architecture with the accuracy set to 0.2 (which is the values used to plot the figures in this section) is equal to the cost of performing 41.3 unblocked matrix-vector products. This means that using the BCSR storage format is convenient only if more than 67 matrix-vector products have to be performed. This number can be substantially reduced decreasing the accuracy of the fill-in estimate. Obviously, reducing the accuracy the fill-in estimate could lead to erroneous block size selection but a good compromise, as suggested in [79], is 0.01. In this case the cost of estimating the fill-in for matrix number 28 on the Itanium2 architecture is just 2.08 and thus only 4 matrix-vector product are enough to absorb the cost of the automatic block size selection [1].

[1] in this case reducing the fill-in estimate accuracy leads to the same block size selection.

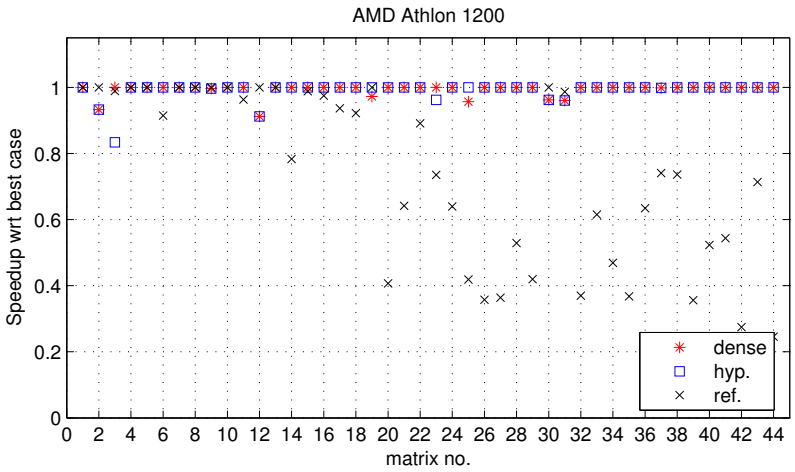

Figure 3.54: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an AMD Athlon 1200 machine.

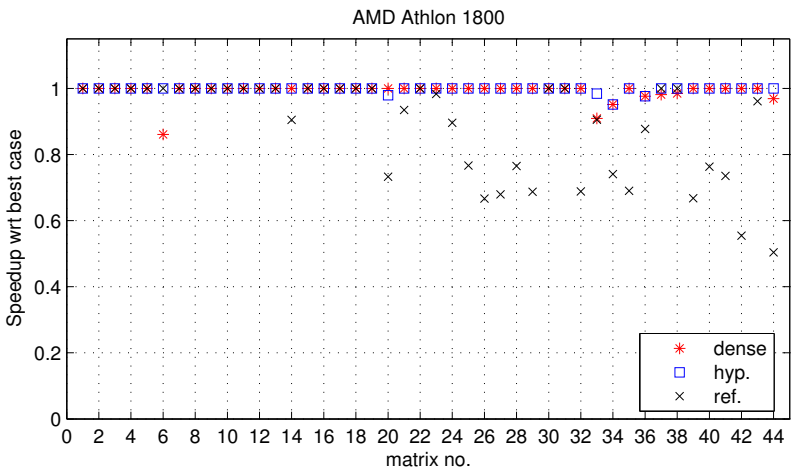

Figure 3.55: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an AMD Athlon 1800 machine.

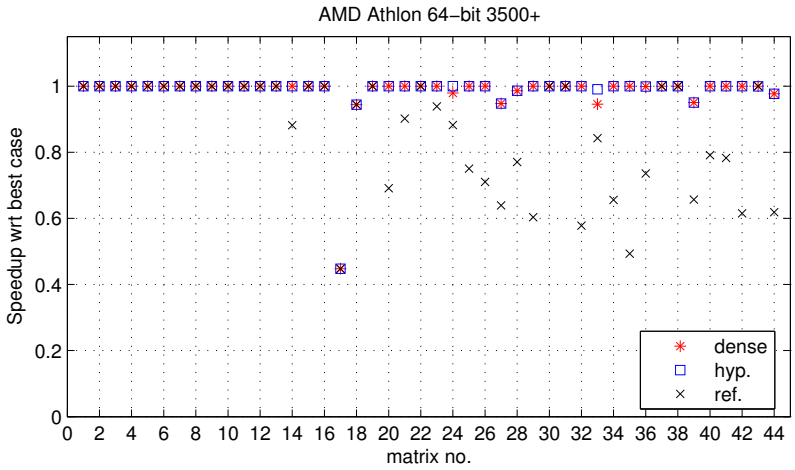

Figure 3.56: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an AMD Athlon 64-bit 3500+ machine.

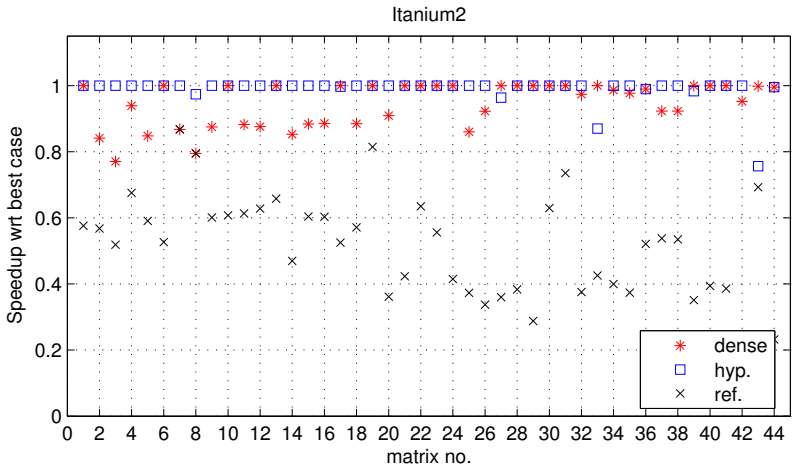

Figure 3.57: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an Itanium2 machine.

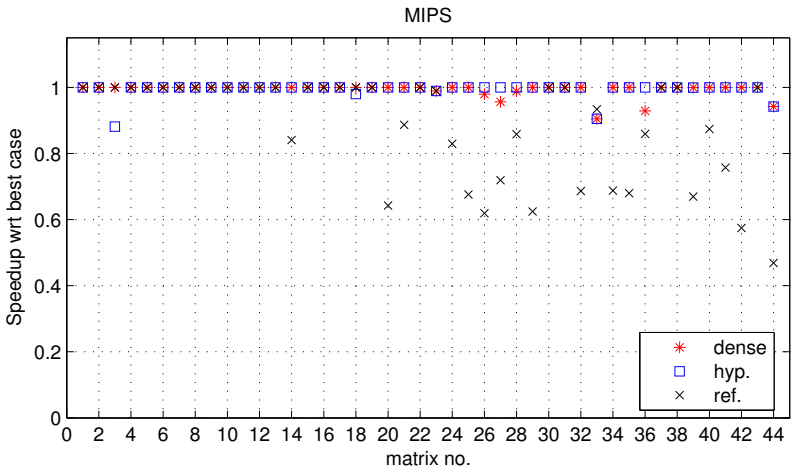

Figure 3.58: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on a MIPS machine.

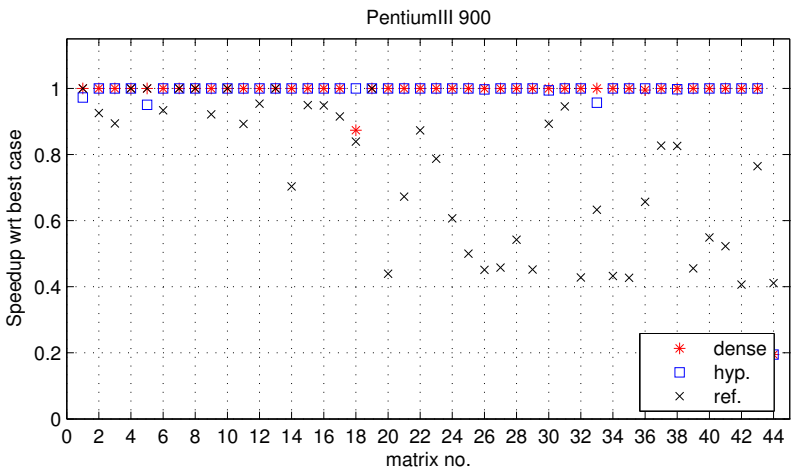

Figure 3.59: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an PentiumIII 900 machine.

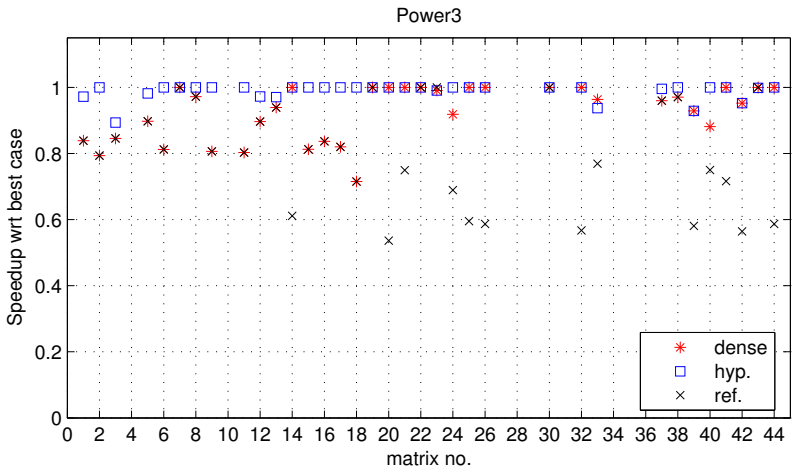

Figure 3.60: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on a Power3 machine.

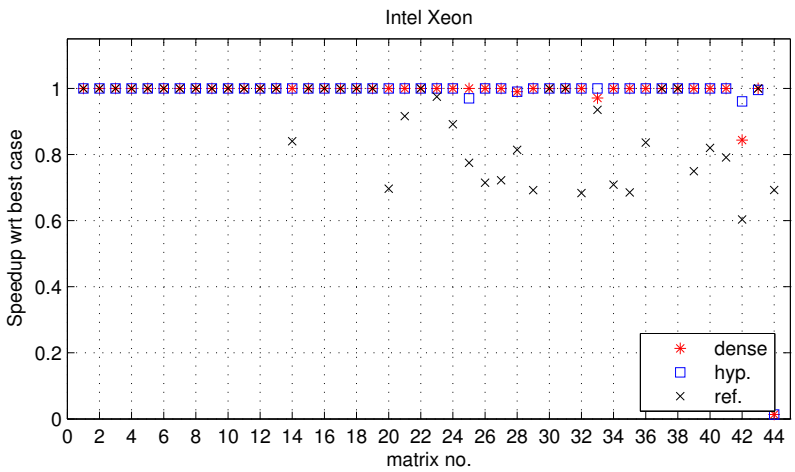

Figure 3.61: Comparison between the time spent in the matrix-vector product operation with the block size selected with the dense matrix based model (*red stars*), the block size selected with the hyperbola based model (*blue squares*) and the CSR case (*black cross*). All the time measures have been normalized to the value of the best-case block size. This data has been measured on an Intel Xeon machine.

Part III

Building Better Algorithms

Chapter 4

Parallel Preconditioners

Contents

4.1 Domain Decomposition Methods

In the context of numerical analysis, *Domain Decomposition Methods* (DDM) refer to a collection of techniques based on the principle of *divide-and-conquer*.

Consider the problem of solving the Laplace equation

$$\begin{cases} \Delta u = f & \text{in } \Omega \\ u = u_\Gamma & \text{on } \Gamma = \partial\Omega \end{cases} \tag{4.1}$$

over the L-shaped domain depicted in figure 4.1. A domain decomposition approach would attempt to solve the problem on the whole domain Ω, by solving the problem on a number of subdomains (three in this case) Ω_i such that:

$$\Omega = \bigcup_{i=1}^{s} \Omega_i$$

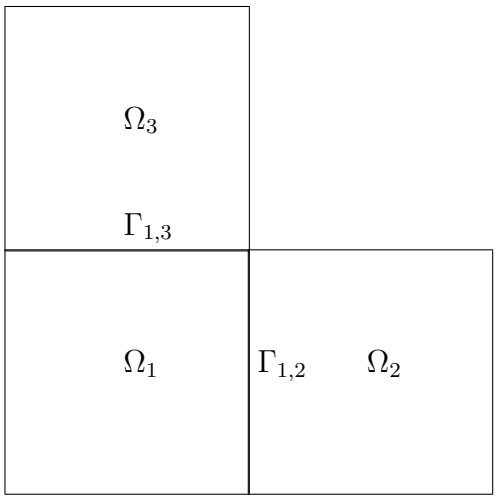

Figure 4.1: An L-shaped domain divided in three subdomains.

where s is the number of subdomains into which the global domain is partitioned.

Domain Decomposition Methods can be considerably advantageous in the cases where it is much easier to solve the problem onto the subdomains because of their simple shape as opposed to the shape of the whole domain, as in the presented example. In other cases it can happen that the physical problem can be split naturally into a small number of subregions where the modeling equations are different. Finally, in parallel programming environments, domain decomposition is often a natural choice because the solution of the problem onto different subdomains can be committed to different processors.

The various domain decomposition techniques are characterized by four features:

1. *type of partitioning*: the way the domain is decomposed. For example on an edge basis or on a vertex basis (discussed in the following).

2. *overlap*: the subdomains need not necessarily be disjoint so how overlapped they are can be an issue. This is the case

$$\Omega = \bigcup_{i=1,s} \Omega_i, \quad \Omega_i \cap \Omega_j \neq \emptyset$$

3. *processing of interface values*: is the Schur complement approach used (not discussed further here. See [64] for more informations)? Should there be successive updates to the interface values?

   Domain decomposition methods are all implicitly or explicitly based on different ways of handling the unknowns at the interfaces.

4. *subdomain solutions*: the problem on each subdomain can be solved exactly or approximately by an iterative method.

This presentation of DDM and parallel preconditioners follows the treatment in [64]; please refer to it and references therein for further details.

4.1.1 Domain Partitioning

There are different ways in which a domain can be partitioned. Two common approaches are edge-based partitioning and vertex-based partitioning. The edge-based technique does not allow edges to be split among two subdomains, i.e. it is not possible to have an edge connecting two nodes in different subdomains. The vertex-based approach is dual to the previous one, i.e. a vertex cannot be split among two (or more) different subdomains. Figures 4.2 and 4.4 show, respectively, and edge-based and a vertex-based partitioning of the domain in figure 4.1. In these two figures also different meshes are used for the discretization of the domain, so, assuming that the problem is discretized with centered differences, different matrices are associated with the two cases (respectively in figure 4.3 and 4.5). The choice of different meshes provides a more balanced partitioning.

In the general case of an edge-based partitioning into s, subdomains, the linear system associated with the problem has the following structure:

$$\begin{pmatrix} B_1 & & & & E_1 \\ & B_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{pmatrix} \tag{4.2}$$

where each x_i represents the subvectors of unknowns that are interior to subdomains Ω_i and y represents the vector of all the interface unknowns. It is useful to express the above system in a simpler form:

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \qquad \text{with} \qquad A = \begin{pmatrix} B & E \\ F & C \end{pmatrix}. \tag{4.3}$$

where E represents the subdomain to interface coupling seen from the subdomains, and F the interface to subdomain coupling seen from the interface nodes.

From the graph theory point of view (which is very useful in domain decomposition techniques), the edge-based partitioning is not as common as the vertex-based one. In the vertex-based partitioning all the edges that

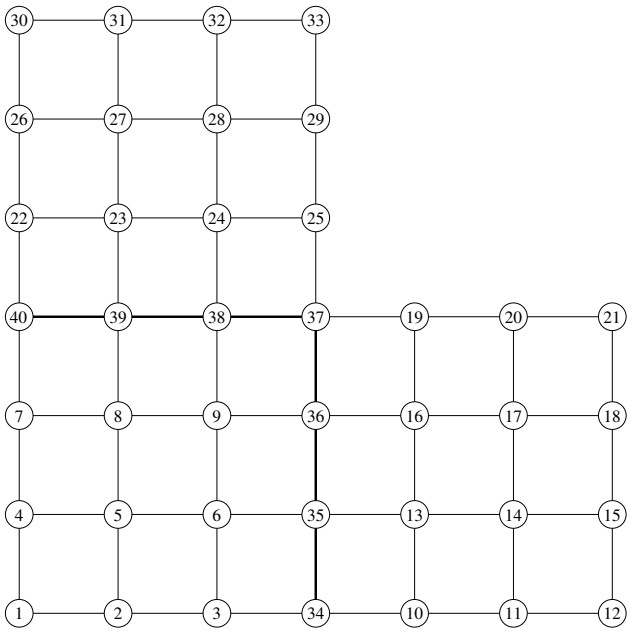

Figure 4.2: Discretization in the problem in figure 4.1 with edge-based domain decomposition.

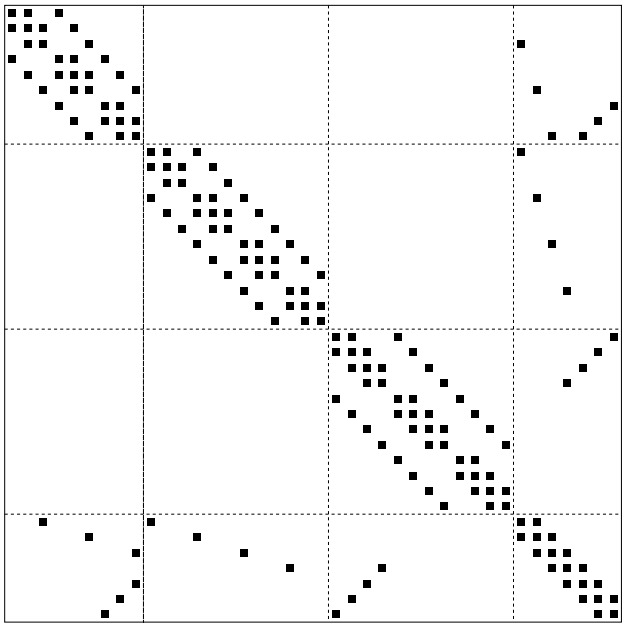

Figure 4.3: Matrix associated with the finite difference mesh in figure 4.2.

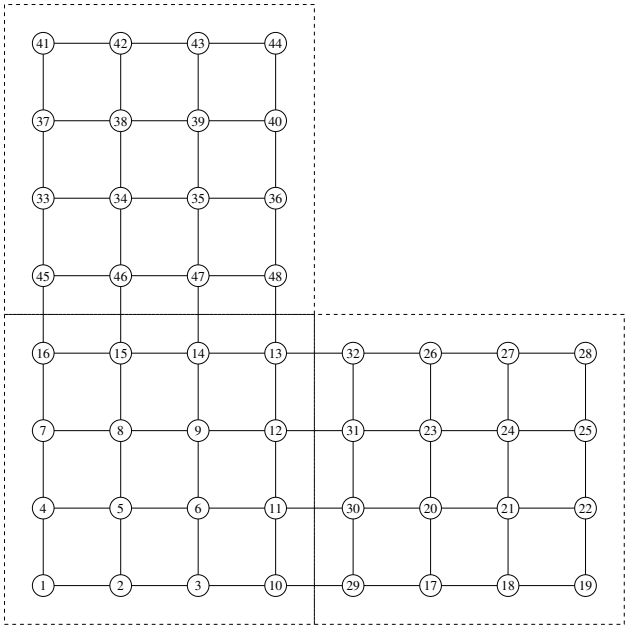

Figure 4.4: Discretization in the problem in figure 4.1 with vertex-based domain decomposition.

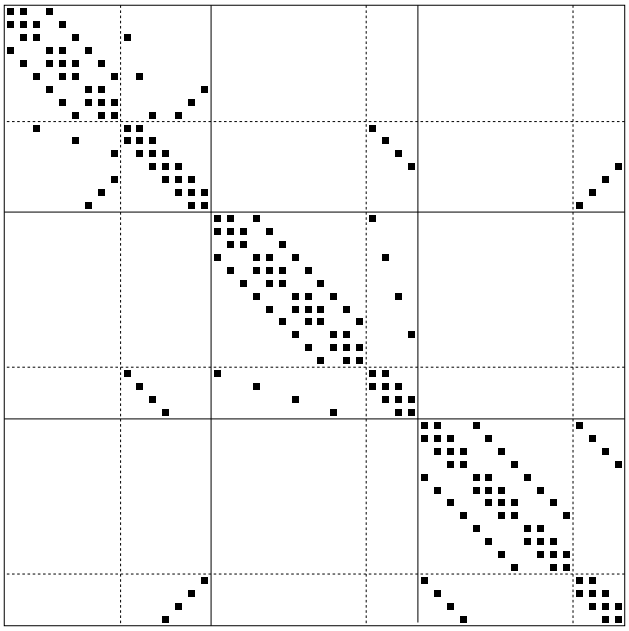

Figure 4.5: Matrix associated with the finite difference mesh in figure 4.4.

connect two nodes that are not in the same subdomain are defined as interface edges. Consider the discretization mesh in figure 4.4 and the associated matrix in figure 4.5.

Note that it would be possible to derive a matrix whose sparsity structure is exactly the same as that of the matrix in figure 4.3 simply by choosing a different number scheme than the one in figure 4.4, namely, a numbering scheme where all the interface nodes are listed last.

However, the numbering proposed in this case makes it easier to exploit the properties of the s-block structure of the matrix depicted in figure 4.5, s being the number of subdomains.

In the case of our example $s = 3$ and so the block structure of the matrix, defined by the solid lines in figure 4.5, can be represented as:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{4.4}$$

In each of the subdomains, the variables are of the form

$$z_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \tag{4.5}$$

where x_i denotes interior nodes and y_i denotes interface nodes associated with subdomain i. Each of the A_{ii} is the local matrix associated with subdomain i and each of them presents the same sparsity structure of the matrix in figure 4.3:

$$A_{ii} = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix} \tag{4.6}$$

in which, as before, B_i represents the matrix associated with the internal nodes of subdomain i, and E_i and F_i represent the coupling to/from external nodes. The submatrix C_i is the local part of the interface matrix C defined before and represents the coupling between local interface points. Each of the A_{ik} blocks, where $i \neq j$, present a zero sub-block in the part that acts on the internal unknowns x_i.

$$A_{ij} = \begin{pmatrix} 0 \\ E_{ij} \end{pmatrix} \tag{4.7}$$

This is because the local unknowns x_i and only coupled with the y_j through E_{ij} but not with the x_j unknowns. Moreover most of the E_{ij} matrices are zero since only those indices j of the subdomains that have couplings with subdomain i will yield a nonzero E_{ij}.

4.2 Additive Schwarz Procedure

The original alternating procedure described by Schwarz in 1870 consisted of three parts:

- alternate between two overlapping domains,
- solve the Dirichlet problem on one domain at each iteration,
- take boundary conditions based on the most recent solution obtained from the other domain.

This procedure is called the "Multiplicative Schwarz" procedure (not discussed further here; see [64]). It is possible to prove that, in some sense, this procedure is equivalent to the block Gauss-Seidel method. The equivalent of the block Jacobi method is the so called "Additive Schwarz" procedure and is the topic of this section.

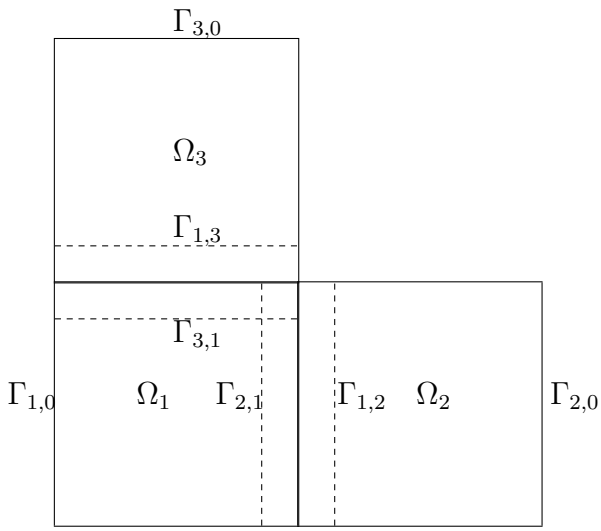

Figure 4.6: An L-shaped domain divided in three overlapping subdomains.

The domain decomposition depicted in figure 4.6, where each pair of neighboring subdomains has a nonvoid overlapping region, will be assumed as a reference in the following discussion. The boundary of subdomain Ω_i that is included in subdomain j is denoted by $\Gamma_{i,j}$. The boundary of Ω_i consisting of its original boundary (denoted with $\Gamma_{i,0}$) and the $\Gamma_{i,j}$ will be called Γ_i and the restriction of the solution u to the boundary $\Gamma j, i$will be called u_{ji}. According to this notation, the original Schwarz Alternating Procedure can be described as in algorithm 9.

Algorithm 9 Schwarz Alternating Procedure

1: Choose an initial guess u to the solution
2: **repeat**
3: **for** $i = 1$ to s **do**
4: Solve $\Delta u = f$ in Ω_i with $u = u_{ij}$ in $\Gamma_{i,j}$
5: Update u values on $\Gamma_{j,i}, \forall j$
6: **end for**
7: **until** Convergence

The algorithm sweeps through the s subdomains and solves the original equation in each of them by using boundary conditions that are updated from the most recent values of u. As already told, this is the Multiplicative Schwarz method and its equivalence with the Gauss-Seidel method can be intuitively explained by the fact that updates are performed as soon as new values are available. Switching to the Additive Schwarz method is simply a matter of moving step 5 of the algorithm outside the inner loop. As in the Jacobi method, then, updates are performed once for all the values at the end of each inner sweep.

Algorithm 10 Additive Schwarz Procedure

1: Choose an initial guess u to the solution
2: **repeat**
3: **for** $i = 1$ to s **do**
4: Solve $\Delta u = f$ in Ω_i with $u = u_{ij}$ in $\Gamma_{i,j}$
5: **end for**
6: Update u values in $\Gamma_{i,j}, \forall i, j$.
7: **until** Convergence

Since each of the subproblem is likely to be solved by an iterative method, it is possible take advantage of a good initial guess. Each of the solutions at step 4 of the algorithm can be translated into an update of the form:

$$u_i := u_i + \delta_i$$

where δ_i, recalling the notation defined in equations 4.4, 4.5 and 4.6, solves the system:

$$A_i \delta_i = r_i$$

Here r_i is the local part of the global residual vector $b - Ax$, and the above system represents the system associated with the problem in step 4 of the

algorithm when a nonzero initial guess is used in some iterative procedure. Writing

$$u_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad \delta_i = \begin{pmatrix} \delta_{x,i} \\ \delta y, i \end{pmatrix}, \quad r_i = \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix}$$

the correction to the current solution step in the algorithm leads to:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}^{-1} \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix} \tag{4.8}$$

The Additive Schwarz procedure can now be formulated as in algorithm 11.

Algorithm 11 Additive Schwarz Procedure - Matrix Form

1: **for** $i = 1$ to s **do**
2: Solve $A_i \delta_i = r_i$
3: **end for**
4: Compute $x_i := x_i + \delta_{x,i}$, $y_i := y_i + \delta_{y,i}$

Algorithm 11 can be easily interpreted in terms of projectors. Let $\mathcal{S}_i$ be an index set

$$\mathcal{S}_i = \{j_1, j_2, ..., j_{n_i}\},$$

where indices j_k are those associated with the n_i mesh points of the interior of the discrete subdomain Ω_i. The $\mathcal{S}_i$'s, which are not necessarily disjoint, form a collection of index sets such that

$$\bigcup_{i=1,...,s} \mathcal{S}_i = 1, ..., n.$$

Let R_i be a *restriction operator* from Ω to Ω_i. By definition, $R_i x$ belongs to Ω_i and keeps only those components of an arbitrary vector x that are in Ω_i.

From the linear algebra point of view, the restriction operator R_i is an $n_i \times n$ matrix formed by the transposes of columns e_j on the $n \times n$ identity matrix where j belongs to the index set $\mathcal{S}_i$. The transpose ${R_i}^T$ of this matrix is a *prolongation operator* which takes a variable from Ω_i end extends it into the equivalent variable in Ω.

The matrices in figure 4.7 represent the three projectors associated with the subdomains in figure 4.4.

The matrix

$$A_i = R_i A {R_i}^T$$

of dimension $n_i \times n_i$ defines a restriction of A to Ω_i.

The basic Additive Schwarz procedure can now be formulated in terms of projectors as in algorithm 12.

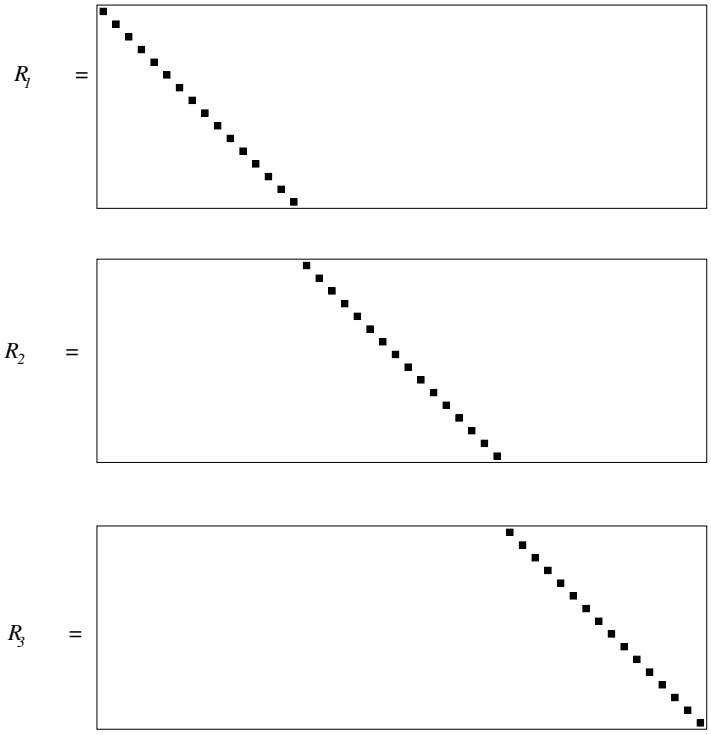

Figure 4.7: The three projectors associated with the subdomains in figure 4.4.

Algorithm 12 Additive Schwarz Procedure

1: **for** $i = 1$ to s **do**
2: Compute $\delta_i = {R_i}^T {A_i}^{-1} R_i(b - Ax)$
3: **end for**
4: Compute $x_{new} = x + \sum_{i=1}{}^{s} \delta_i$

The new approximation (obtained after a cycle of s substeps in algorithm 12) is:

$$x_{new} = x + \sum_{i=1}^{s} {R_i}^T {A_i}^{-1} R_i(b - Ax)$$

Each instance of the loop redefines different components of the new approximation. It is important to note that there is no data dependency between the subproblems involved in the loop and thus, this procedure, is characterized by a high degree of parallelism. Because of the equivalence of the Additive Schwarz procedure and a block Jacobi iteration, it is possible to recast one Additive Schwarz sweep in the form of a global fixed-point iteration of the form $x_{new} = Gx + f$. This is a fixed-point iteration for solving a preconditioned

system $M^{-1}Ax = M^{-1}b$ where the preconditioning matrix M and the matrix G are related by $G = I - M^{-1}A$.

The preconditioning matrix is rather simple to obtain for the Additive Schwarz procedure. The new iterate satisfies the relation:

$$x_{new} = x - \sum_{i=1}^{s} T_i(b - Ax) = \left(I - \sum_{i=1}^{s} P_i \right) x + \sum_{i=1}^{s} T_i b$$

where $P_i = {R_i}^T {A_i}^{-1} R_i A$ and $T_i = P_i A^{-1}$. From the last formula it is possible to obtain:

$$G = I - \sum_{i=1}^{s} P_i. \qquad f = \sum_{i=1}^{s} T_i b$$

With the relation $G = I - M^{-1}A$ between G and the preconditioning matrix M, the result is that

$$M^{-1}A = \sum_{i=1}^{s} P_i$$

and

$$M^{-1} = \sum_{i=1}^{s} P_i A^{-1} = \sum_{i=1}^{s} T_i$$

Now the procedure for applying the preconditioning operator becomes the one outlined in algorithm 13.

Algorithm 13 Additive Schwarz Preconditioner

1: Input:v; Output: $z = M^{-1}v$
2: **for** $i = 1$ to s **do**
3: Compute $z_i := T_i v$
4: **end for**
5: Compute $z = z_1 +, ..., + z_s$

Note that the do loop can be performed in parallel because also in this case there is no data dependency between the loops. Step 5 sums up the vectors z_i in each domain to obtain a global vector z. In the non-overlapping case this step i parallel and consists of just forming this different components since the addition is trivial.

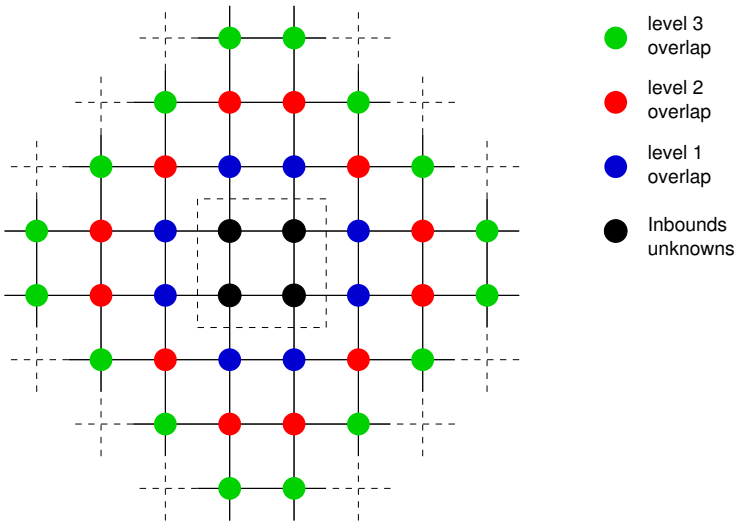

Figure 4.8: Overlap levels.

4.2.1 Algebraic Schwarz Algorithms

Consider a linear system

$$Ax = b \tag{4.9}$$

where $A = (a_{ij})$ is an $n \times n$ nonsingular sparse matrix having a nonzero pattern that is symmetric. Let $G = (W, E)$ be a graph where the set of vertices $W = \{1, ..., n\}$ represents the n unknowns and the edge set $E = \{(i, j)|a_{ij} \neq 0\}$ represents the pairs of vertices that are coupled by a nonzero element in A. Assuming that A has a symmetric nonzero pattern, the adjacency graph G is undirected. Two vertices are considered to be neighbors if there is an edge connecting them. Assume that a graph partitioning has been applied to G and has resulted in m nonoverlapping subsets W_i^0 such that $\bigcup_{i=1}^m W_i^0 = W$. This is called a *0-overlap* partition of W. A *δ-overlap* partition of W with $\delta > 0$ can be defined recursively by considering the sets $W_i^\delta \supset W_i^{\delta-1}$ obtained by including the vertices that are neighbors of the vertices in $W_i^{\delta-1}$ as in figure 4.8.

Let n_i^δ be te size of W_i^δ and R_i^δ the $n_i^\delta \times n$ matrix formed by the row vectors e_j^T of the $n \times n$ identity matrix, with $j \in W_i^\delta$. For each $v \in \mathbb{R}^n$, $R_i^\delta v$ is the vector containing the components of v corresponding to the vertices in W_i^δ, hence R_i^δ can be viewed as a restriction operator from $\mathbb{R} =^n$ to $\mathbb{R}^{n_i^\delta}$. Likewise, the transpose matrix $(R_i^\delta)^T$ can be viewed as a prolongation operator from $\mathbb{R}^{n_i^\delta}$ to $\mathbb{R} =^n$. The *Additive Schwarz (AS)* preconditioner, M_{AS}, is then

defined by

$$M_{AS}^{-1} = \sum_{i=1}^{m} (R_i^\delta)^T (A_i^\delta)^{-1} R_i^\delta$$

where the $n_i^\delta \times n_i^\delta$ matrix $A_i^\delta = R_i^\delta A (R_i^\delta)^T$ is obtained by considering the rows and columns of A corresponding to the vertices in W_i^δ. Note that although A_i is not invertible, it is possible to invert its restriction to the subspace

$$A_i^{-1} \equiv ((A_i)_{|L_i})^{-1}$$

where L_i is the vector space spanned by the set W_i^δ in $\mathbb{R}^n$.

When $\delta = 0$, M_{AS} is the well-known Block Jacobi preconditioner. The convergence theory for the AS preconditioner is well developed in the case of symmetric positive definite matrices (see [20] and references therein). It can be proven that, when the AS preconditioner is used in conjunction with a Krylov subspace method, the convergence rapidly improves as the overlap δ increases, while it deteriorates as the number m of subsets W_i^δ grows. Theoretical results are available also in the case of nonsymmetric and indefinite problems.

Recently some variants of the AS preconditioner have been developed that outperform the classical AS for a large class of matrices, in terms of both convergence rate and of computation and communication time on parallel distributed-memory computers [16, 32, 42]. In particular, the *Restricted Additive Schwarz (RAS)* preconditioner, M_{RAS}, and the *Additive Schwarz* preconditioner *with Harmonic extension (ASH)*, M_{ASH}, are defined by

$$M_{RAS}^{-1} = \sum_{i=1}^{m} (\tilde{R}_i^0)^T (A_i^\delta)^{-1} R_i^\delta, \qquad M_{ASH}^{-1} = \sum_{i=1}^{m} (R_i^\delta)^T (A_i^\delta)^{-1} \tilde{R}_i^0,$$

where $\tilde{R}_i^0$ is the $n_i^\delta \times n$ matrix obtained by zeroing the rows of R_i^δ corresponding to the vertices in $W_i^\delta \backslash W_i^0$. The application of the AS preconditioner requires the solution of m independent linear systems of the form

$$A_i^\delta w_i = R_i^\delta v \tag{4.10}$$

and then the computation of the sum

$$\sum_{i=1}^{m} (R_i^\delta)^T w_i. \tag{4.11}$$

In the RAS preconditioner, R_i^δ in 4.11 is replaced by $\tilde{R}_i^0$; hence, in a parallel implementation where each processor holds the rows of A with indices

in W_i^0 and the corresponding components of right-hand side and solution vectors, this sum does not require any communication. Analogously, in the ASH preconditioner, R_i^δ in equation 4.10 is replaced by $\tilde{R}_i^0$; therefore, the computation of the right-hand side does not involve any data exchange among processors.

In the applications, the exact solution of system 4.10 is often prohibitively expensive. Thus, it is customary to substitute the matrix $(A_i^\delta)^{-1}$ with an approximation $(K_i^\delta)^{-1}$, computed by incomplete factorizations, such as ILU, or by iterative methods, such as SSOR or Multigrid (see [20]).

4.3 Building and Applying AS Preconditioners in PSBLAS

This section reviews the basic operations involved in the Additive Schwarz preconditioners from the point of view of parallel implementation through PSBLAS routines. What follows is based on the distinction between

preconditioner setup: the set of basic operations needed to identify W_i^δ, to build A_i^δ from A, and to compute K_i^δ from A_i^δ;

preconditioner application: the set of basic operations needed to apply the restriction operator R_i^δ to a given vector v, to compute (an approximation of) w_i, by applying $(K_i^\delta)^{-1}$ to the restricted vector, and, finally, to obtain sum 4.11.

Existing PSBLAS computational routines implement the operations needed for the application phase of AS preconditioners, provided that a representation of the δ-partition is built and packaged into a new suitable data structure during the phase of preconditioner setup. The next two sections are devoted to these points.

4.3.1 PSBLAS implementation of preconditioner application

To compute the right-hand side in 4.10 the restriction operator R_i^δ must be applied to a vector v distributed among parallel processes conformally to the sparse matrix A. On each process the action of R_i^δ corresponds to gathering the entries of v with indices belonging to the set $W_i^\delta \backslash W_i^0$. This is the semantics of the PSBLAS `psb_halo` routine, which updates the halo components of a vector, i.e. the components corresponding to the 1-overlap indices. The same code can apply an arbitrary δ-overlap operator, if a suitable auxiliary descriptor data structure is provided.

Likewise, the computation of sum 4.11 can be implemented through a suitable call to the PSBLAS computational routine `psb_ovrl`; this routine can compute either the sum, the average or the square root of the average of the vector entries that are replicated in different processes according to an appropriate descriptor.

Finally, the computation of $(K_i^\delta)^{-1}v_i^\delta$, where $v_i^\delta = R_i^\delta v$ or $v_i^\delta = \tilde{R}_i^0 v$, can be accomplished by two calls to the sparse block triangular solve routine `psb_spsm`, given a local (incomplete) factorization of A_i^δ.

Therefore, the functionalities needed to implement the application phase of the AS, RAS and ASH preconditioners, in the routine `psb_precaply`, are provided by existing computational PSBLAS routines, if an auxiliary descriptor `psb_desc_type` (see section 2.2.2) is built. Thus, the main effort in implementing the preconditioners lies in the definition of a preconditioner data structure and of routines for the preconditioner setup phase, as discussed in Section 4.3.2.

4.3.2 PSBLAS implementation of preconditioner setup

The implementation of the AS preconditioners is based on the definition of the `psb_prec_type` data structure (the one presented in section 2.2.4 that has been successively extended to implement Multi-Level preconditioners) that includes in a single entity all the items involved in the application of the preconditioner (refer to `psb_base_prec` data structure in figure 2.4):

- a preconditioner identifier and the number δ of overlap layers in `iprcparm(:)`;
- two sparse matrices in `av(:)`, holding the lower and upper triangular factors of K_i^δ (the diagonal of the upper factor is stored in a separate array, `d(:)`);
- the auxiliary descriptor `desc_data`, built from the sparse matrix `A`, according to the number of overlap levels.

Note that the sparse matrix descriptor is kept separate from the preconditioner data; with this choice the sparse matrix operations needed to implement a Krylov solver are independent of the choice of the preconditioner.

Algorithm 14 outlines the procedure to setup an instance of the `psb_prec_type` structure for AS, RAS or ASH, with overlap `n_ovr`. By definition the submatrices A_i^0 identify the vertices in W_i^1; the relevant indices are stored into the initial communication descriptor. Given the set W_i^1, process i may request the column indices of the non-zero entries in the

rows corresponding to $W_i^1 \backslash W_i^0$; these in turn identify the set W_i^2, and so on. All the communication is performed in the steps 6 and 10, while the other steps are performed locally by each process. A new auxiliary routine, `psb_asbld`, has been developed to execute the steps 1–10. To compute the triangular factors of K_i^δ (step 11), the existing PSBLAS routine `psb_spilu`, performing an ILU(0) factorization of A_i^δ, is currently used. The two previous routines have been wrapped into a single PSBLAS application routine, named `psb_asbld`.

It would be possible to build the matrices A_i^δ while building the auxiliary descriptor `desc_data`. Instead, the two phases have been separated, thus providing the capability to reuse the `desc_data` component of an already computed preconditioner; this allows efficient handling of common application situations where multiple linear systems with the same structure must be solved.

Algorithm 14 Preconditioner Setup Algorithm

1: Initialize the descriptor `desc_data` by copying the matrix descriptor `desc_a`.
2: Initialize the overlap level: $\eta = 0$.
3: Initialize the local vertex set, $W_i^\eta = W_i^0$, based on the current descriptor.

4: **while** $\eta <$ n_ovr **do**
5: Increase the overlap: $\eta = \eta + 1$.
6: Build W_i^η from $W_i^{\eta-1}$, by adding the halo indices of $A_i^{\eta-1}$, and exchange with other processors the column indices of the non-zero entries in the rows corresponding to $W_i^\eta \backslash W_i^{\eta-1}$.
7: Compute the halo indices of A_i^η and store them into `desc_data`.
8: **end while**
9: **If** (`n_ovr` > 0) Optimize the descriptor `desc_data` and store it in its final format.
10: Build the enlarged matrix A_i^δ, by exchanging rows with other processors.

11: Compute the triangular factors of the approximation K_i^δ of A_i^δ.

4.4 Numerical Experiments

The Additive Schwarz preconditioning implementations discussed above have been tested with a number of matrices coming from real world applications whose properties are described in table 4.1. The preconditioners, built for

0, 1 and 2 overlap levels[1], are applied as right preconditioners with the BiCGSTAB solver available in PSBLAS, choosing the null vector as starting guess. The iterations are stopped when the ratio between the 2-norms of the residual and of the right-hand-side is less than 10^{-10}; a maximum number of 7000 iterations is also set, but it is never reached in the tests. A row-block distribution of the matrices is used, where each processor holds (approximately) equal-sized blocks of consecutive rows, according to the well-known BLACS one-dimensional pure-block mapping. A conformal distribution is applied to the right-hand side and solution vectors. This choice implicitly defines a domain decomposition such that the number of subdomains is equal to the number of processors.

The tests discussed here have been carried out on a cluster with dual-processor nodes, installed at the Innovative Computing Laboratory of the University of Tennessee at Knoxville. Each node has an AMD Opteron dual-processor (model 240, 1.4 GHz), running the Debian Linux 3.1 operating system with kernel 2.6.13, and 2 GBytes of memory; the nodes are connected with Myrinet network interfaces. The tests have been run on 32 nodes, i.e. on 64 processors. A development snapshot of the GNU compilers version 4.2, including both the C and Fortran 95 compilers, was used in conjunction with the specific MPI implementation for the Myrinet interface.

Before going on with discussing the measured data, it is worth to remark that theoretical results have been developed that show why Restricted Additive Schwarz preconditioners tend to perform better than classical Additive Schwarz ones (see [32] for details). These results are only partially applicable to de case discussed in the present chapter; the reason lies in the fact that ILU(0) incomplete factorization is used in PSBLAS to solve the preconditioning system (1.18) yielding considerably different results with respect to the case where the preconditioning system is exactly solved. Even if the obtained results show that, in most of the cases, the Restricted Additive Schwarz preconditioning behave better, in an algebraic sense, there are problems, like the "memplus" one (see table 4.1, where using the classical Additive Schwarz preconditioner may yield significant improvements.

Besides the number of iterations it is also important to remark that the cost of each single iteration is not the same among the different Additive Schwarz preconditioning implementations. In this sense, the RAS and ASH preconditioners are less expensive than the classical AS one because, at time when the preconditioner is applied, the prolongation and restriction

[1]higher overlap level have not been analyzed for reason that will be explained in the following considerations. Remark, also, that every AS preconditioner where the number of overlap levels is 0, is perfectly equivalent to the Block Jacobi one.

Matrix	Size	nnz	comment
add32	4960×4960	19848	32-bit adder model
language	399130×399130	1216334	Finite-state machines for natural language processing
memplus	17758×17758	91147	Memory circuit model
poisson3Db	13514×13514	352762	Test matrices from FEMLAB, a finite-element method toolbox for MATLAB
kivap1	86304×86304	1575568	Automitove engine design
therm2D	600000×600000	2996800	Steady-state thermal diffusion equation

Table 4.1: Properties of the matrices used to test Additive Schwarz preconditioners.

operations, respectively, are not performed yielding a reduction of the time per iteration.

Figure 4.9 show that Additive Schwarz preconditioning (regardless of the variant) provides a strong reduction in the number of iterations on the matrices described in table 4.1. Except for the "memplus" matrix, where the classical AS turns out to be the most efficient, the Restricted AS provides the strongest reduction in the number of iterations. For each matrix only the most efficient variant of the AS preconditioning is reported in figure 4.9 for the sake of space.

Figure 4.9 also shows that, in general, it possible to say that the number of iterations decreases as the number of overlapping levels grows. Matrix "therm2D" shows an almost irregular behavior in this sense if compared to the other matrices in the testset. However it is important to note that, as the number of overlap levels increases, the amount of communication to be performed at preconditioner application phase also increases. This translates in a higher cost per iteration which is possible to see in table 4.2. Thus a relatively small gain in the number of iterations may still yield higher solver times due to the increased cost of the preconditioner application.

Figure 4.10 shows the overhead of building an Additive Schwarz preconditioner compared to the time spent in solving the system. As it is possible to see, the cost of building the preconditioner grows with the number of overlap level.

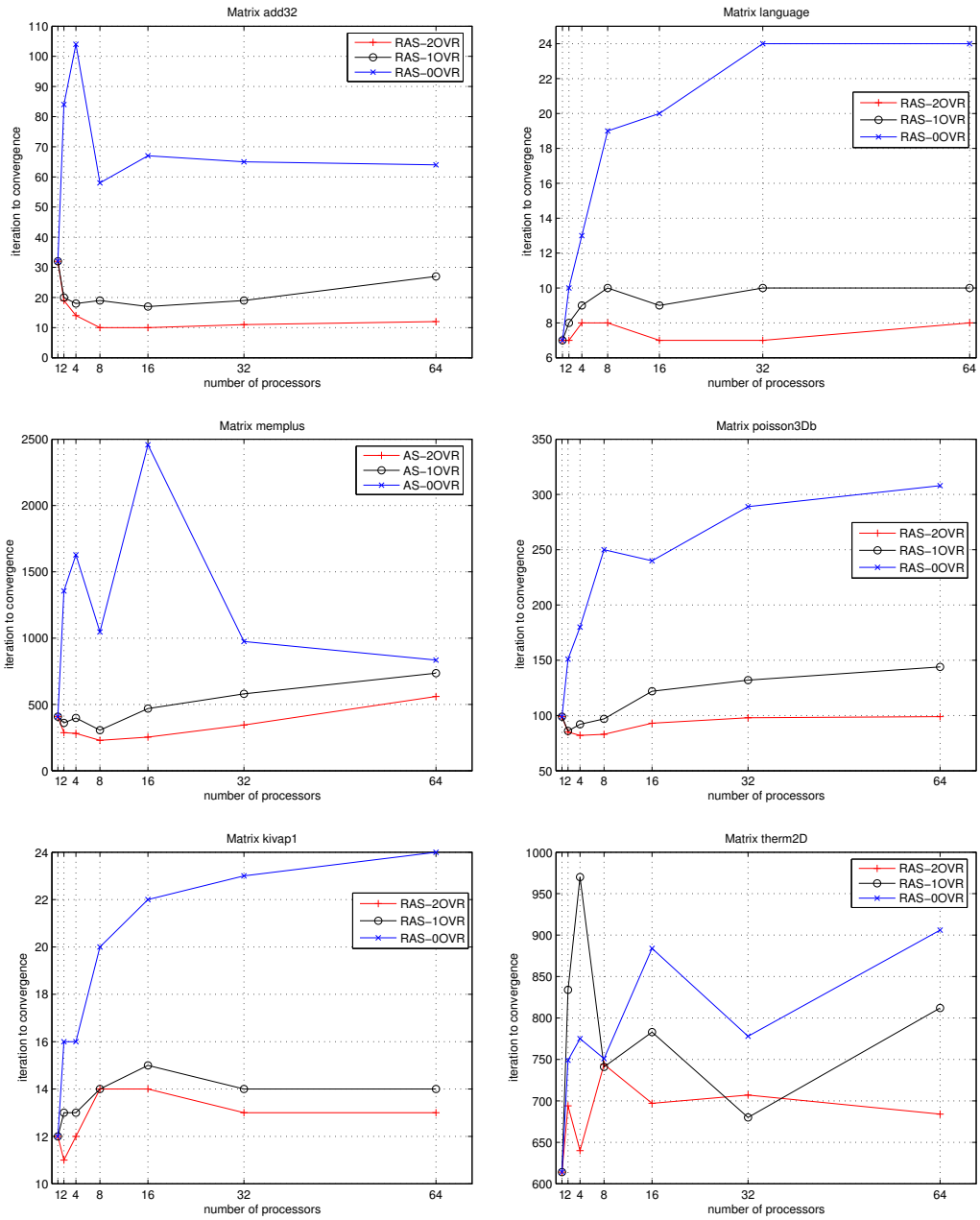

Figure 4.9: Number of iterations versus number of processes for 0, 1 and 2 overlapping levels.

add32									
	RAS-0			RAS-1			RAS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	32	0.0027	0.0521	32	0.0029	0.0622	32	0.0029	0.0636
2	84	0.0012	0.1285	20	0.0114	0.0393	19	0.0180	0.0376
4	104	0.0008	0.0833	18	0.0167	0.0389	14	0.0250	0.0311
8	58	0.0004	0.0537	19	0.0110	0.0297	10	0.0167	0.0165
16	67	0.0002	0.0402	17	0.0063	0.0183	10	0.0101	0.0108
32	65	0.0002	0.0314	19	0.0047	0.0152	11	0.0079	0.0092
64	64	0.0001	0.0281	27	0.0054	0.0185	12	0.0102	0.0107

language									
	RAS-0			RAS-1			RAS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	7	0.1354	1.3910	7	0.1237	1.5330	7	0.1238	1.5200
2	10	0.0730	1.1610	8	0.4216	1.2150	7	0.8012	1.2800
4	13	0.0449	0.9331	9	0.4106	1.1630	8	0.8747	1.3850
8	19	0.0243	0.7583	10	0.3638	0.7348	8	0.7687	0.8381
16	20	0.0157	0.4660	9	0.2289	0.3817	7	0.5824	0.5049
32	24	0.0116	0.3323	10	0.1507	0.2746	7	0.4722	0.3732
64	24	0.0094	0.2383	10	0.1178	0.1979	8	0.3688	0.3270

memplus									
	AS-0			AS-1			AS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	408	0.0766	3.6770	408	0.0769	3.9810	408	0.0769	4.0060
2	1356	0.0799	8.4520	361	0.1134	3.8500	287	0.1779	3.6780
4	1628	0.0396	9.7260	398	0.1380	5.6220	282	0.2136	4.8620
8	1047	0.0238	4.5750	306	0.1404	3.0070	229	0.2721	3.9530
16	2458	0.0100	6.8890	469	0.1368	4.0600	254	0.3193	4.1710
32	975	0.0060	2.9180	580	0.1522	4.3600	345	0.3544	5.5570
64	835	0.0047	2.6110	736	0.1648	5.6520	560	0.4858	9.8790

poisson3Db									
	RAS-0			RAS-1			RAS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	99	1.3200	16.2900	99	1.3210	16.2200	99	1.3190	16.0700
2	151	0.3582	12.3700	83	1.6900	11.1600	83	2.8150	11.4800
4	180	0.1013	11.3900	80	2.1320	13.0800	86	4.3270	14.4000
8	250	0.0466	10.1100	83	2.2370	10.3300	81	4.8140	10.7300
16	240	0.0224	7.3080	80	2.4740	9.6710	73	5.0240	10.6800
32	289	0.0134	6.4220	82	2.1280	7.7350	76	4.1530	9.0860
64	308	0.0078	5.4590	96	1.6650	6.7270	80	4.3310	11.3100

kivap1									
	RAS-0			RAS-1			RAS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	12	0.2661	1.2040	12	0.2668	1.2520	12	0.2667	1.2500
2	16	0.1296	0.7108	13	0.1613	0.6258	11	0.2030	0.5694
4	16	0.0635	0.3712	13	0.1170	0.3474	12	0.1884	0.3662
8	20	0.0311	0.2279	14	0.0789	0.2046	14	0.1484	0.2565
16	22	0.0157	0.1330	15	0.0579	0.1281	14	0.1144	0.1601
32	23	0.0081	0.0779	14	0.0528	0.0828	13	0.1069	0.1137
64	24	0.0044	0.0388	14	0.0446	0.0610	13	0.0972	0.0848

therm2D									
	RAS-0			RAS-1			RAS-2		
np	it	Tset.	Tslv.	it	Tset.	Tslv.	it	Tset.	Tslv.
1	614	0.2201	172.8000	614	0.2212	187.5000	614	0.2214	187.4000
2	749	0.1185	108.0000	834	0.1229	130.0000	694	0.1254	116.5000
4	775	0.0623	54.9800	970	0.0682	75.0900	640	0.0771	54.5300
8	751	0.0348	26.9800	741	0.0410	29.4700	744	0.0505	32.1800
16	884	0.0214	15.2900	783	0.0285	15.2000	697	0.0375	14.8000
32	778	0.0140	6.7590	680	0.0220	6.7170	707	0.0315	7.7370
64	906	0.0113	5.0670	812	0.0225	5.6390	684	0.0337	5.4020

Table 4.2: Additive Schwarz preconditioners measured performance in terms of number of iterations, preconditioner setup time and solver time.

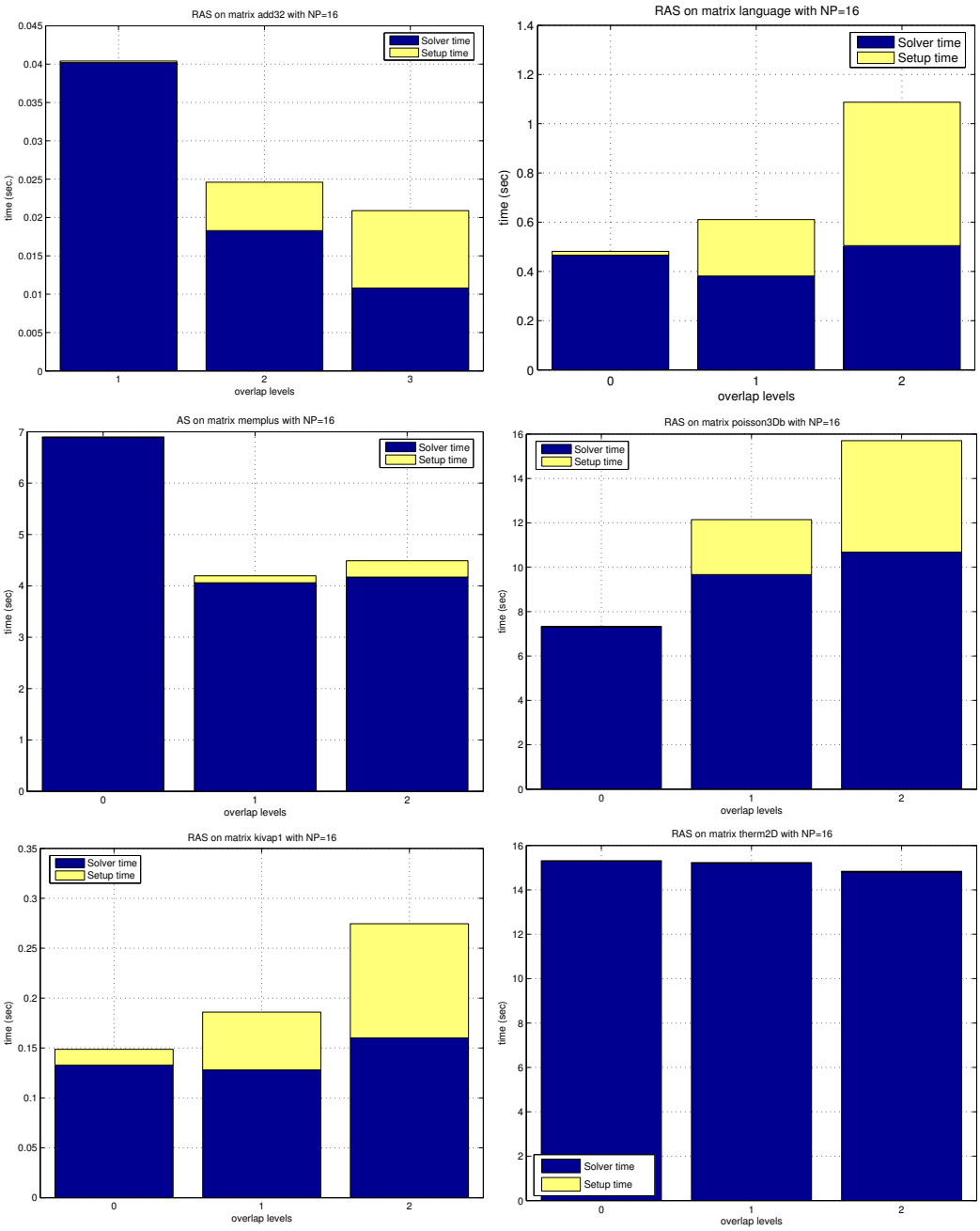

Figure 4.10: Additive Schwarz preconditioner setup and system solver times for 0, 1 and 2 overlap levels.

This is fully explained by the fact that when the number of overlap levels grows:

1. building the preconditioner communication descriptor becomes more expensive,

2. the amount of communications performed when building the A^{δ} matrix increases,

3. the a^{δ} matrix size also increases making it more expensive to compute its ILU(0) factorization.

It is important to note that there are cases where, even if the solver time is reduced as the number of overlap levels grows, the total time may still be higher due to the overhead of building the AS preconditioner. This, for example, happens for the "language" and "kivap1" matrices: moving from 0 to 1 overlap levels, the solver time is reduced but the overall computation time is higher due to the building phase cost. Anyway even in these cases, using Additive Schwarz preconditioners may still yield some improvement because:

- the same preconditioner may be used more than once like in applications where the same matrix has to be solved against many right hand sides,

- if two matrices have the same sparsity structure, then the associated AS preconditioners will also share the same sparsity structure abd communication descriptor; inthese cases it is possible to reuse topological information on all subsequent preconditioner build steps.

Figure 4.11 plots the solver time for the AS preconditioners versus the number of processors for 0, 1 and 2 overlap levels. This figure shows that AS preconditioned solvers have good scalability properties except for the "poisson3Db" and "memplus" matrices where it is not possible to identify a regular behavior. Figure 4.11 also shows that Additive Schwarz preconditioning is not suited for all the matrices. In the case of the "add32" matrix, Additive Schwarz preconditioning provides good speedups with a considerably regular behavior while in other cases, like for example, the "poisson3Db" matrix the Block Jacobi preconditioner (equivalent to the AS preconditioner with 0 overlap levels) outperforms it. For the "memplus" matrix, the applicability of AS preconditioning methods depends on the number of processors used providing some improvement when the number of processors is lower than 32. Finally there are also matrices, like, the "language" one where improvements are obtained only when the number of overlap levels is limited.

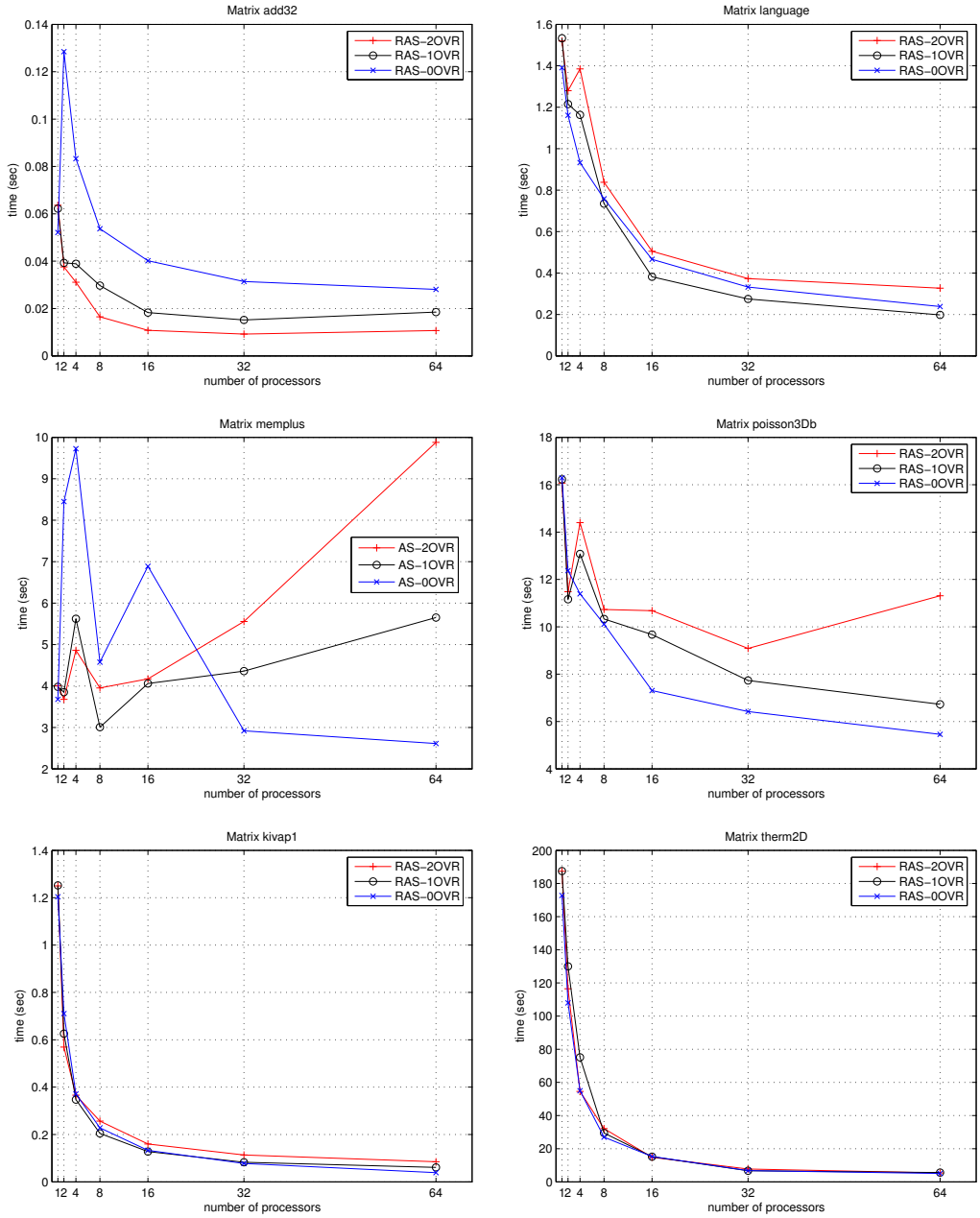

Figure 4.11: Solver time versus number of processes for 0, 1 and 2 overlapping levels.

Chapter 5

Multigrid Preconditioners

Contents

5.1 Multigrid Methods

In this section a brief introduction to multigrid methods is presented which will be successively extended to multigrid preconditioners in the following section.

Multilevel methods, extensively discussed in [80], [65] and [69], have been developed to improve the convergence behavior of iterative linear systems solvers. Roughly speaking, common stationary iterative solvers, like the Gauss-Seidel or the Jacobi ones, may be affected by poor convergence rates due to the fact that they are point solvers, meaning that the algebraic equation is solved on each point of the grid independent of the solution at

any other point[1]. As a result, such methods, allow a very quick reduction of local errors while obtaining poor results in reducing global errors. This means that for errors whose length scales are comparable to the grid mesh size, they provide rapid damping, leaving behind smooth, longer wave-length errors; these smooth components are responsible for the slow global convergence. Figure 5.1 shows how error is smoothed through few Gauss-Seidel iterations: high frequency local error are reduced while long wave stay almost unchanged causing low convergence rate.

A multigrid method attempts to improve global convergence employing a number of grids with different mesh sizes aiming at reducing all wavelength errors.

The multigrid strategy combines two complementary schemes. The high-frequency components of the error are reduced applying iterative methods like Jacobi or Gauss-Seidel schemes. For this reason these methods are called smoothers. On the other hand, low-frequency error components are effectively reduced by a coarse-grid correction procedure. Because the action of a smoothing iteration leaves only smooth error components, it is possible to represent them as the solution of an appropriate coarser system. Once this coarser problem is solved, its solution is interpolated back to the fine grid to correct the fine grid approximation for its low-frequency errors.

[1]Other iterative techniques attempt to add some portion of a direct solver to the mix. line solvers are a popular example of this. In a line solver, an entire "line" of points are solved coupled, points off the line are not coupled. Line solvers are powerful, due to the point coupling, but are difficult to implement on unstructured grids.

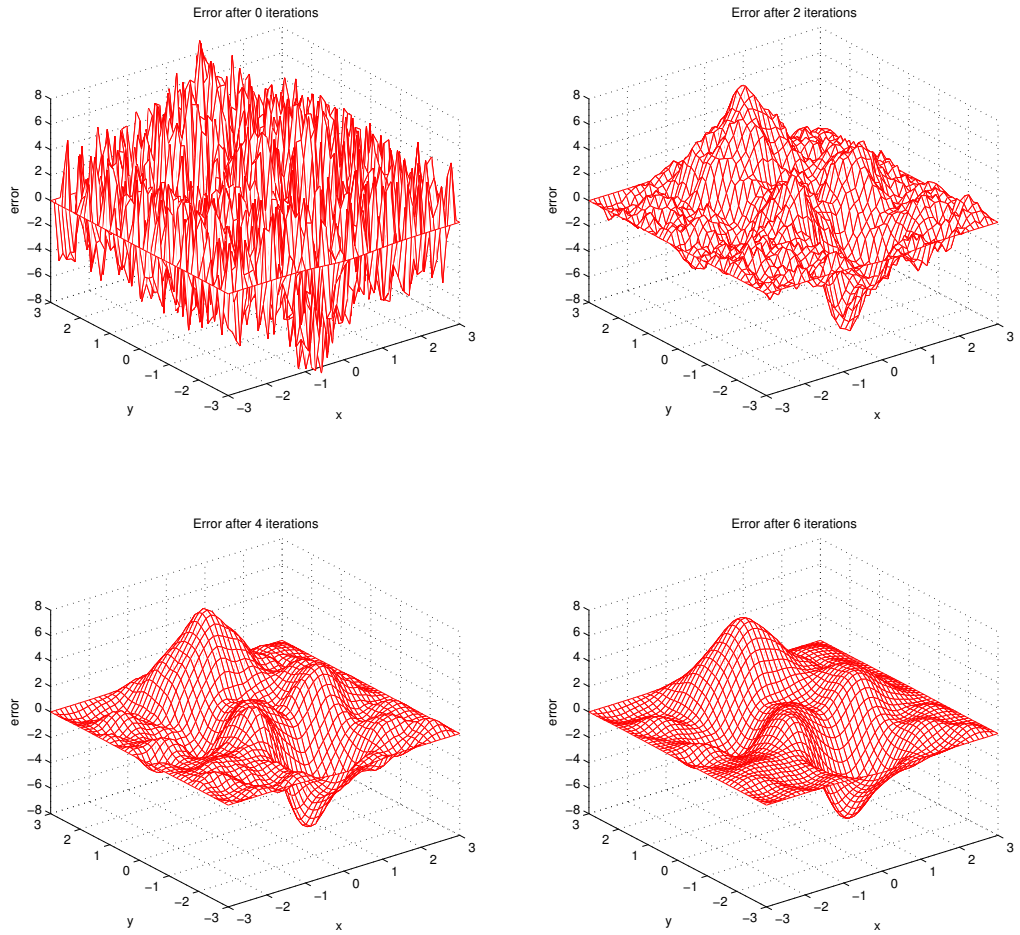

Figure 5.1: A sample of how error is reduced through few iterations of the Gauss-Seidel iterative method. It is possible to see how local errors are rapidly damped while long wave error are almost not reduced.

5.1.1 Iterative Methods and the smoothing property

Consider a system $Ax = b$ where A is a $n \times n$ matrix. As already discussed in section 1.1, stationary iterative methods may be expressed in the form:

$$x^{(k+1)} = Mx^{(k)} + Nb \tag{5.1}$$

where M and N have to be constructed in such a way that the sequence $x^{(k)}, k = 1, ...$ converges to the solution $x = A^{-1}b$ in a finite number of steps. Defining the error at step k is defined as $e^{(k)} = x - x^{(k)}$, equation 5.1 can be

rewritten as:

$$e^{(k+1)} = Me^{(k)} \tag{5.2}$$

where M is the so-called *iteration* matrix whose characteristics determine the convergence behavior of the iterative method. Theorem 1 (in section 1.1) states that the method converges to the system solution if $\rho(M) < 1$, i.e. if the spectral radius of the iteration matrix is lower than one.

Many common methods are built based on the *splitting* technique. The system matrix can be splitted as $A = B - C$ where B is non-singular; setting $Bx^{(k+1)} + Cx^{(k)} = b$ and solving with respect to $x^{(k+1)}$ a stationary iterative method can be formulated as:

$$x^{(k+1)} = B^{-1}Cx^{(k)} + B^{-1}b \tag{5.3}$$

In this case, obviously, $M = B^{-1}C$ and $N = B^{-1}$. The splitting $A = D - L - U$, where $D = diag(a_{11}, a_{22}, ..., a_{nn})$ and $-L$ and $-U$ are respectively the lower triangular and upper triangular parts of A, leads to the well known (damped-)Jacobi and Gauss-Seidel methods.

The damped-Jacobi method is derived defining $B = \frac{1}{\omega}D$ and $C = \frac{1}{\omega}[(1 - \omega)D + \omega(L + U)]$ which yields:

$$x^{(k+1)} = (I - \omega D^{-1}A)x^{(k)} + \omega D^{-1}b, \tag{5.4}$$

while the Gauss-Seidel method comes from the choice $B = D - L$ and $C = U$:

$$x^{(k+1)} = (D - L)^{-1}Ux^{(k)} + (D - L)^{-1}b \tag{5.5}$$

The iteration matrices for these two methods will be referred as $M_J(\omega)$ and M_{GS}.

To analyze the smoothing property of an iterative solver, consider a one-dimensional Dirichlet boundary value problem:

$$\begin{cases} -\frac{d^2u}{dx^2} &= f(x), \quad in \quad \Omega = (0,1) \\ u(x) &= g(x), \quad on \quad x \in 0,1 \end{cases} \tag{5.6}$$

Let Ω be represented by a grid Ω_h with grid size $h = 1/(n+1)$ and points $x_h = jh, j = 0, 1, 2, ..., n+1$. Assuming $f_j^h = f(jh)$ and $u_j^h = u(jh)$, the discretization can be performed with the following scheme for the second order derivative at x_j:

$$\frac{1}{h^2}\left[u(x_{j-1}) - 2u(x_j) + u(x_{j+1})\right] = u''(x_j) + O(h^2)$$

Thus, the following system of n equation is obtained:

$$\left\{ \begin{array}{rcl} \frac{2}{h^2}u_1^h - \frac{1}{h^2}u_2^h & = & f_1^h + \frac{1}{h^2}g(0) \\ -\frac{1}{h^2}u_{j-1}^h + \frac{2}{h^2}u_j^h - \frac{1}{h^2}u_{j+1}^h & = & f_j^h, \qquad j = 2, ..., n-1 \\ -\frac{1}{h^2}u_{n-1}^h + \frac{2}{h^2}u_n^h & = & f_n^h + \frac{1}{h^2}g(1) \end{array} \right. \tag{5.7}$$

Consider the Jacobi method and the associated eigenvalue problem $M_J(\omega)v^k = \mu_k v^k$. The eigenvectors are given by:

$$v^k = (sin(k\pi hj))_{j=1,n}, \qquad k = 1, ..., n \tag{5.8}$$

and the corresponding eigenvalues:

$$\mu_k(\omega) = 1 - \omega(1 - cos(k\pi h)), \qquad k = 1, ..., n \tag{5.9}$$

Equation 5.9 shows that convergence is guaranteed only in the case where $0 < \omega \leq 1$ that implies $\rho(M_J(\omega)) < 1$.

Assuming $\omega = 1$ the spectral radius of the iteration matrix is:

$$\rho(M_J(1)) = \max_k |cos(\pi kh)| \tag{5.10}$$

$$= \max_k |1 - \frac{1}{2}(\pi kh)^2 + O(h^4)| \tag{5.11}$$

$$= 1 - \frac{1}{2}(\pi h)^2 + O(h^4) \tag{5.12}$$

showing how convergence behavior deteriorates with $h \to 0$. In order to define the Jacobi method smoothing property the following distinction can be done:

- **low frequency error modes:** eigenvectors v^k with $1 \leq k < \frac{n}{2}$;
- **high frequency error modes:** eigenvectors v^k with $\frac{n}{2} \leq k < n$.

Considering that the error can be written as a linear combination of the iteration matrix eigenvectors, i.e. $e^{(k)} = \sum_i e_i^{(k)}$, then it is possible to state that at each Jacobi iteration, an error component is damped by a value which is equal to the corresponding eigenvalue.

Choosing, for example, $\omega = 2/3$ and the eigenvalues have the values plotted in figure 5.2. It is straightforward from this figure to understand how the error modes are damped at each Jacobi step; high frequency components are reduced by at least 70%, low frequency components are reduced at best by 70% while very low frequency error modes are barely reduced.

Due to this property stationary iterative solvers, in multigrid methods, are used as smoothers to reduce high frequency error modes; low frequency error modes are, instead, reduced applying a coarse grid correction as explained in the next section.

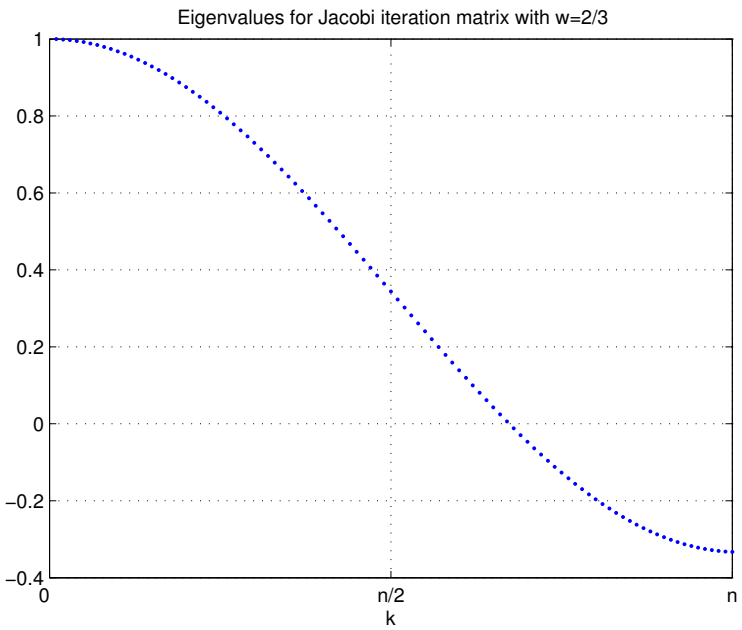

Figure 5.2: The eigenvalues of the damped-Jacobi iteration matrix with $\omega = 2/3$

5.1.2 Multi-Grid scheme

As already discussed in the previous section, a method like Jacobi is highly effective at killing off high-frequency error components. On the other hand, slow convergence is due to slow reduction of low-frequency components. The notion of high and low frequency is, however, relative to the grid spacing. Consider, for example, two grids for the problem (5.6); a grid with h spacing and a coarser one with $H = 2h$ spacing, respectively the black and the red grid in figure 5.3. So on grid H, some of what looked like low frequency errors on grid h now look more like high frequency. Multigrid is based on the idea that both high and low frequency error components can be damped combining partial solutions of the problem on grids with different spacings.

After the application of few smoothing steps (Jacobi sweeps for example) an approximation of the solution $\tilde{x}_h$ is obtained whose error is $\tilde{e}^h = x^h - \tilde{u}^h$[2]. Then $\tilde{e}^h$ can be approximated on a coarser space. This smooth error must be expressed as the solution of a coarser problem whose matrix A_H and

[2]the subscript h is used to refer to the problem on grid with h spacing as opposed to the problem on grid with $H = 2h$ spacing

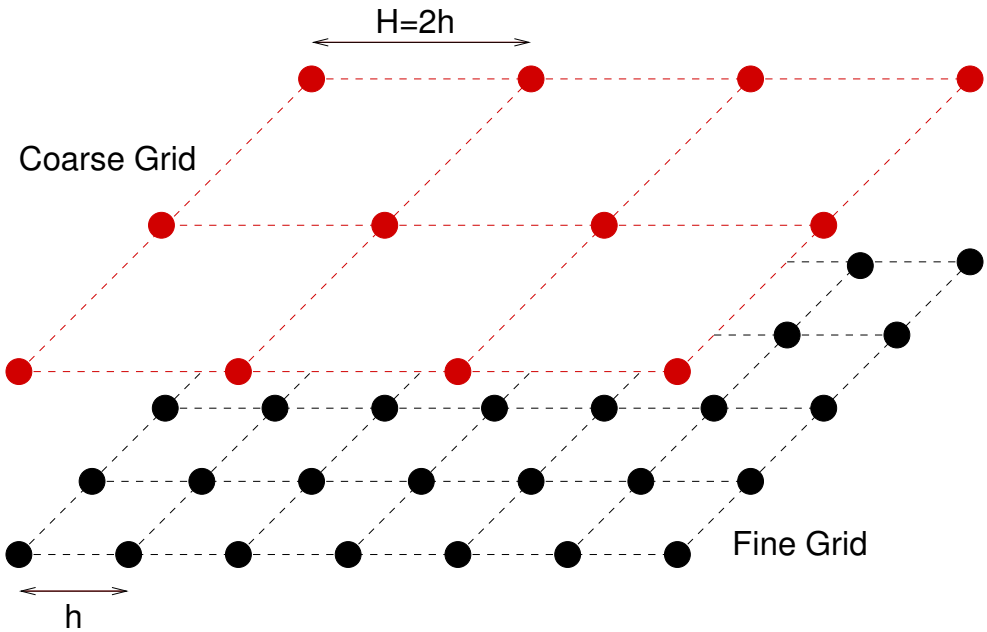

Figure 5.3: Two grids with different mesh sizes for the same domain. The black fine mesh has mesh size h while the red coarse one has mesh size $H = 2h$.

right-hand side have to be defined. Notice that in problem (5.6) A^h is a second order difference operator and $r^h = b^h - A^h\tilde{x}^h$ is a smooth function if $\tilde{e}^h$ is smooth. Obviously the original equation $A^h x^h = b^h$ and the residual equation $A^h\tilde{e}^h = r^h$ are equivalent. The difference between the two equations is that $\tilde{e}^h$ and r_h are smooth, therefore it is possible to represent them on a coarser grid with spacing $H = 2h$. Define r^H as the restriction of the fine grid residual on the coarse grid, that is, $r^H = I_h^H r^h$ where I_h^H is a suitable restriction operator. This defines the right-hand side of the coarse problem. Since $\tilde{e}^h$ is the solution of a difference operator which can be represented analogously on the coarse discretization level, the following coarse problem ca be defined:

$$A^H\tilde{e}^H = r^H \tag{5.13}$$

with homogeneous Dirichlet boundary conditions as for $\tilde{e}^h$. Here A^H represents the same discrete operator but relative to the grid with mesh size H. It is reasonable to expect that $\tilde{e}^H$ is an approximation of $\tilde{e}^h$ on the coarse grid. Inversely, $\tilde{e}^H$ can be brought back to the fine space with a prolongation operator I_H^h. Therefore, since by definition $x^h = \tilde{x}^h + \tilde{e}^h$ the solution $\tilde{x}^h$ can

be updated applying the following coarse grid correction step :

$$\tilde{x}^h_{new} = \tilde{x}^h + I^h_H \tilde{e}^H. \tag{5.14}$$

In practice, also the interpolation procedure may introduce high frequency errors on the fine grid problem and, thus, it could be convenient to perform some more smoothing steps after the coarse grid correction has been applied. Given that $x^{h^{(l)}} = Mx^{h^{(l-1)}} + Nb^h$ is a smoothing procedure, it will be denoted as $x^{h^{(l)}} = S^h(x^{h^{(l-1)}}, b^h)$ in the following.

Algorithm 15 Two-Grid Method

1: **for** $l = 1$ to ν_1 **do**
2: apply the pre-smoothing step $x^{h^{(l)}} = S^h(x^{h^{(l-1)}}, b^h)$
3: **end for**
4: compute the residual $r^h = b^h - A^h x^{h^{(\nu_1)}}$
5: restrict the residual $r^H = I^H_h r^h$
6: solve the coarse-grid problem $e^H = (A^H)^{-1} r^H$
7: apply the coarse-grid correction $x^{h^{(\nu_1+1)}} = x^{h^{(\nu_1)}} + I^h_H e^H$
8: **for** $l = 1$ to ν_2 **do**
9: apply the post-smoothing step $x^{h^{(l)}} = S^h(x^{h^{(l-1)}}, b^h)$
10: **end for**

Algorithm 15 shows the pseudo-code for a two-grid procedure which can be iterated until convergence to the system solution; a two-grid scheme starts at the fine level with pre-smoothing, performs a coarse-grid correction solving a coarse-grid auxiliary problem, and ends with post-smoothing. This is the so-called V-cycle.

In the two-grid scheme previously discussed, the size of the coarse grid mesh is twice as large as the fine grid mesh size; this means that the coarse problem can still be very large making unpractical to solve it directly as described in algorithm 15. However the coarse problem has the same form as the residual problem on the fine level and, thus, the two-grid method can be used to determine $\tilde{e}^H$; so equation 5.13 can be itself solved by two-grid cycling introducing a further coarse grid problem.

For a more detailed description define a sequence of grids with mesh size $h_1 > h_2 >, ..., > h_L < 0$ so that $h_k = 2h_{k-1}, k = 1, 2, ..., L$. Let Ω^{h_k} denote the set of grid points with grid spacing h_k; the number of interior grid points is n^k. On each level k a problem $A^k x^k = b^k$ can be defined where A^k is an $n^k \times n^k$ matrix and x^k and b^k are vectors of size n^k. The transfer among levels is performed with I^{k-1}_k restriction and I^k_{k-1} prolongation operators. The

Algorithm 16 Multi Grid Method

```
if k = 1 then
    solve A^k x^k = b^k directly
end if
for l = 1 to ν_1 do
    apply the pre-smoothing step x^{k(l)} = S^k(x^{k(l-1)}, b^k)
end for
compute the residual r^k = b^k − A^k x^{k(ν_1)}
restrict the residual r^{k-1} = I_k^{k-1} r^k
set x^{k-1} = 0
call γ times the MG method to solve A^{k-1} x^{k-1} = r^{k-1}
apply the coarse-grid correction x^{k(ν_1+1)} = x^{k(ν_1)} + I_{k-1}^k x^{k-1}
for l = 1 to ν_2 do
    apply the post-smoothing step x^{k(l)} = S^k(x^{k(l-1)}, b^k)
end for
```

smoothing iteration is denoted by $x^k = S^k(x^k, b^k)$. The multigrid method can now be expressed as in

The γ parameter is the number of times the MG method is applied to the coarser level problem. Since this procedure converges fast, $\gamma = 1$ or $\gamma = 2$ (in which case the multigrid scheme is called W-cycle) are typical values.

5.1.3 Algebraic Multigrid

Multigrid methods have been generally defined in the previous section as methods operating on a hierarchy of grids obtained by coarsening the given discretization mesh in a geometrically natural way. For this reason, the approach previously presented is defined as "geometric" multigrid and it is almost straightforward to understand and implement for logically regular grids. However defining a coarsening strategy to build a grid hierarchy may be very complicated, if possible at all, when using highly complex unstructured meshes.

First attempts to automate the extraction of coarser grids consist of combining geometric multigrid with *Galerkin-principle* and *operator-dependent interpolation*. This attempt, presented in [11], was motivated by the fact that the Galerkin operator and the operator-dependent interpolation may be derived directly from the underlying matrix without any reference to the discretization mesh. A detailed description of Algebraic MultiGrid (AMG), as this approach has been called, is presented in [63] and [69] where this discussion has been extracted from.

A two-level AMG cycle to solve s.p.d systems of equations

$$A^h x^h = b^h, \qquad or \qquad \sum_{j \in \Omega^h} a^h_{ij} x^h_j = b^h_i \quad (i \in \Omega^h) \tag{5.15}$$

is formally defined as a geometric two-grid cycle. The only difference is that, in the context of AMG, Ω^h is just an index set, while it corresponds to a grid in geometric multigrid. Accordingly, on a coarser level, Ω^H is defined as a smaller index set.

In classical AMG, coarse-level variables are regarded as a subset of fine-level ones. The set of fine-level variables is, thus, split into two disjoint subsets $\Omega^h = C^h \cup F^h$, with C^h representing those variables which are in the coarse-level and F_h being the complementary set. Given such a C/F-splitting, the set of coarse level variables is $\Omega^H = C^h$ and interpolations $e^h = e^H I^h_H$ can be defined as:

$$e^h_i = (e^H I^h_H)_i = \begin{cases} e^H_i & if \quad i \in C^h \\ \sum_{k \in P^h_i} w^h_{ik} e^H_k & if \quad i \in F^h \end{cases} \tag{5.16}$$

where $P^h_i \subset C^h$ is called the set of *interpolatory variables*.

While geometric multigrid is essentially based on the usage of robust smoothers, AMG assumes the usage of simple relaxation schemes, like Gauss-Seidel or Jacobi, and then attemtps to build a suitable interpolation I^h_H. From now on the superscripts used to identify the level will be omitted, where possible, to improve readability. As reported in [69], it can be heuristically assumed that $|r_i| << a_{ii}|e_i|$ which implies that the error e_i can be locally approximated by the following formula:

$$e_i \approx -(\sum_{j \in N_i} a_{ij} e_j)/a_{ii} \quad (i \in \Omega) \tag{5.17}$$

where $N_i = \{j \in \Omega : j \neq i, a_{ij} \neq 0\}$ denoted the *neighborhood* of any point $i \in \Omega$. Such an error is called *algebrically smooth*. The goal of AMG is to define C/F-splittings and sets of interpolatory variables $P_i \subset C(i \in F)$ along with corresponding weights w_{ik} such that equation (5.16) yields a reasonable approximation for each algebrically smooth error e.

Choosing $P_i = N_i$ and $w_{ik} = -a_{ik}a_{ii}$ obviously yields good results but it basically means that each of the points $i \subset F$ has all of its neighbors in C. This choice is practically unattractive since it poses very high computational and memory requirements. In practice it is important to reduce the set of interpolatory variables in order to simplify the coarsening process and to have sparse Galerkin operators in a way to keep the convergence behavior as good as possible.

In the case where the system matrix A is an M-matrix the method proposed in [63] results to be particularly efficient. This method is based on the observation that an algebrically smooth error varies slowly in the direction of strong couplings[3]. The more strong couplings of any variable i are contained in P_i, the better an algebrically smooth error satisfies:

$$\frac{1}{\sum_{k\in P_i} a_{ik}} \sum_{k\in P_i} a_{ik} e_k \approx \frac{1}{\sum_{j\in N_i} a_{ij}} \sum_{j\in N_i} a_{ij} e_j \quad (i \in \Omega). \tag{5.18}$$

Using the previous approximation inside equation (5.16) the following "direct[4] interpolation" is obtained:

$$w_{ik} = -\alpha_i \frac{a_{ik}}{a_{ii}} \quad where \quad \alpha_i = \frac{\sum_{j\in N_i} a_{ij}}{\sum_{l\in P_i} a_{il}} \tag{5.19}$$

Roughly speaking, this means that C/F-splittings have to be built in such a way that each variable $i \in F$ has a sufficiently large number of strongly coupled C neighbors which are then taken as the set of interpolatory variables P_i.

Another way of building C/F-splitting and interpolations is by "aggregation". This is a very simple method where each F-variable interpolates from exactly one C-variable only. These kind of interpolation doesn't yield good convergence behavior[5], but, its ease of implementation and its low computational requirements make it attractive in some cases like, for example, when multigrid methods are used for preconditioning purposes. In aggregation-type AMG C/F-splittings and interpolations are built considering $w_{ik} = 1$ for just one $k \in C$ and zero otherwise for each $i \in F$. Consequently the total number of variables can be subdivided in aggregates $I_k (k \in C)$ where I_k contains (apart from itself) all indices i corresponding to F-variables which interpolate from variable k as

This implies:

$$I_h^H A_h I_H^h = (A_{kl}^H) \quad where \quad a_{kl}^H = \sum_{i\in I_k} \sum_{j\in I_l} a_{ij}^h \quad (k, l \in C) \tag{5.20}$$

that is, the coefficient a_{kl}^H is just the sum of all cross-couplings between I_k and I_l. A sophisticated way to accelerate the basic aggregation-type AMG

[3] a point i is said to be strongly connected to j if $-(a_{ij}) \geq \theta_0 \cdot \max_{k\neq i}(-a_{ik}), \quad 0 < \theta_0 \leq 1$

[4] an interpolation is called direct if $P_i \subset C$

[5] better results are achieved in the case where each F-variable is surrounded by its C interpolatory variables. For a detailed discussion of these properties see [69]

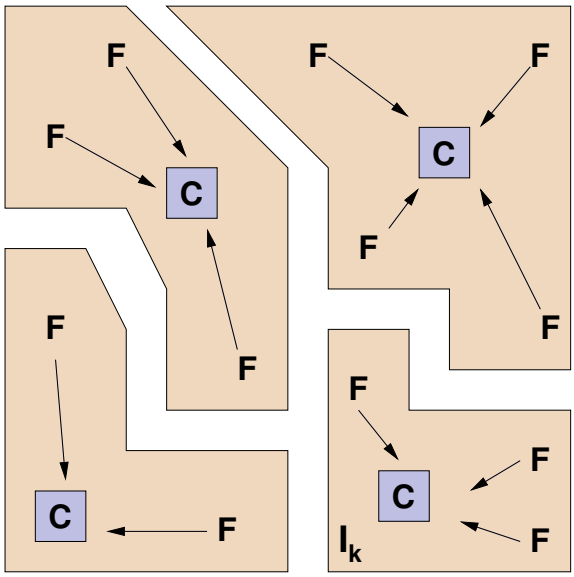

Figure 5.4: Coarsening by aggregation sample. The arrows show how each F-variable interpolates from only one C-variable.

is based on a *smoothing process* used to improve the piecewise constant interpolation which is only used as a first-guess interpolation ("smoothed aggregation"). This smoothing procedure consists in the application of one damped-Jacobi step. Denote the operator corresponding to the piecewise constant interpolation by $\tilde{I}_H^h$. Then the interpolation actually used is defined by:

$$I_H^h = (I_h - \omega D_h^{-1} A_h^f)\tilde{I}_H^h \tag{5.21}$$

where $D_h = diag(A_h^f)$ and A_h^F is derived from the original matrix A_h by adding all weak connections to the diagonal. This means that, given some coarse level vector e^H, $e^h = I_H^h e^H$ is defined by applying one ω-Jacobi relaxation step to the homogeneous $A_f^h v^h = 0$, starting with the piecewise constant operator $\tilde{I}_H^h e^H$.

In general, classical AMG and AMG based on smoothed aggregation perform comparably if applied to relatively smooth (Poisson-like) problems. A considerable advantage of (smoothed) aggregation-type AMG is that in most of the cases it poses lower memory requirements with respect to classical AMG due to its very fast coarsening which determines a very low operator complexity. On the other hand such methods result to be less robust than classical AMG when used as stand-alone solvers. The robustness of AMG multigrid methods can be improved combining them with acceleration methods

such as CG, or Bi-CGSTAB or other Krylov-subspace family solvers. This development was driven by the observation that it is often not only simpler but also more efficient to use accelerated multigrid methods rather than using efficient but also very complex multigrid stand-alone solvers. Moving our point of view on iterative solvers, this actually means using multigrid methods for system preconditioning. As shown in next sections, this approach results to be very efficient in many cases; in fact, even if it poses much higher computational requirements than, say, one-level ILU-type preconditioners, the number of iterations may be drastically reduced providing, then, lower solution times.

5.2 Two-level Schwarz Preconditioners

Domain Decomposition (DD) preconditioners presented in section 4.1, coupled with Krylov iterative solvers, are widely used in the parallel solution of sparse linear systems arising from large-scale applications. Results presented in section 4.4 show that Additive Schwarz preconditioners provide good convergence properties and, in some cases, considerable reduction in the time needed for the solution of a system. Moreover their intrinsic parallelism make them a favorable choice in many cases. A further improvement to this family of preconditioners may come from the use of some extra *coarse space* where the original linear system can be approximated to provide optimal convergence rates, i.e. a number of iterations independent of the number of subdomains [17, 65].

This is the base of two-level, and more generally multilevel, DD preconditioners. Since on parallel computers the number of subdomains usually matches the number of available processors, the use of a coarse space is necessary for developing scalable preconditioning algorithms.

When the matrix to be preconditioned is explicitly related to the geometry of some problem, e.g. when a structured PDE discretization is considered, building such a coarse space may be fairly easy. However, in a pure algebraic formulation, the construction of a coarse space is not a trivial task. Two basic strategies can be identified: *classical Algebraic Multigrid coarsening* [63] and *aggregation* [12, 77] also discussed in section 5.1.3. As already explained in previous sections, in the former case, the set of vertices of the adjacency graph of the matrix is partitioned into two disjoint subsets, one including the vertices that are related to either the fine or the coarse space, the other including the vertices that are only in the fine space and will be interpolated by using coarse-space vertices. In the latter, the coarse-space vertices are obtained as aggregates of fine-space vertices. In both cases, the

coarsening algorithms have a sequential nature. Several parallel versions of these algorithms have been developed, aimed at building a coarse space that realizes a tradeoff between good convergence properties and low communication costs. Details on this issue are given in [31] and in the references therein. The following sections describe the implementation of Two-Level Additive Schwarz preconditioners presented in [14].

5.2.1 Definition of two-level preconditioners

Two-level Schwarz preconditioners are obtained by combining Schwarz preconditioners with a coarse-space correction step.

Recall from section 4.2.1 that a one-level AS preconditioner is defined by

$$M_{1L}^{-1} = \sum_{i=1}^{m} (\tilde{R}_i^{\delta_1})^T (A_i^{\delta})^{-1} \tilde{R}_i^{\delta_2}, \tag{5.22}$$

where A_i^{δ} is assumed to be nonsingular. The classical AS preconditioner corresponds to $\tilde{R}_i^{\delta_j} = R_i^{\delta}$ $(j = 1, 2)$; the Restricted AS (RAS) preconditioner corresponds to $\delta_1 = 0$ and $\delta_2 = \delta$, where $\tilde{R}_i^0 \in \Re^{n_i^{\delta} \times n}$ is obtained by zeroing the rows of R_i^{δ} identified by the vertices in $W_i^{\delta} \backslash W_i^0$; finally, the AS preconditioner with Harmonic extension (ASH) correspond to $\delta_1 = \delta$ and $\delta_2 = 0$. Also remember that in actual applications it is customary to compute an approximate solution of $w_i^{\delta} = (A_i^{\delta})^{-1} v_i^{\delta}$, e.g. by using an incomplete factorization of A_i^{δ} or a basic iterative method such as SOR.

The convergence rate of the one-level Schwarz preconditioned iterative solvers deteriorates as the number m of partitions of W increases [65]. To reduce the dependency of the number of iterations on the degree of parallelism it is possible to introduce a global coupling among the partitions by defining a coarse-space approximation A_C of the matrix A. In a pure algebraic setting, A_C is usually built with a Galerkin approach. Given a set W_C of *coarse vertices*, with size n_C, and a suitable restriction operator $R_C \in \Re^{n_C \times n}$, A_C is defined as[6]

$$A_C = R_C A R_C^T \tag{5.23}$$

and the coarse-space correction matrix to be combined with the one-level AS preconditioners is obtained as

$$M_C^{-1} = R_C^T A_C^{-1} R_C, \tag{5.24}$$

where A_C is assumed to be nonsingular.

[6] according to the notation previously used, the restriction operator is $R_C = I_h^H$ while the prolongation one is $R_C^T = I_H^h$.

The combination of M_C and M_{1L} can be performed in either an additive or a multiplicative framework. In the former case, the *two-level additive* Schwarz preconditioner is obtained:

$$M_{2L-A}^{-1} = M_C^{-1} + M_{1L}^{-1}. \tag{5.25}$$

Applying M_{2L-A}^{-1} to a vector $v \in \Re^n$ corresponds to applying M_C^{-1} and M_{1L}^{-1} to v independently and then summing up the results.

In the multiplicative case, the combination can be performed either by computing

$$\begin{aligned} w &= M_{1L}^{-1} v, \\ z &= w + M_C^{-1}(v - Aw), \end{aligned} \tag{5.26}$$

which corresponds to the following *two-level hybrid* Schwarz preconditioner

$$M_{2L-H1}^{-1} = M_C^{-1} + \left(I - M_C^{-1} A\right) M_{1L}^{-1}, \tag{5.27}$$

or by computing

$$\begin{aligned} w &= M_C^{-1} v, \\ z &= w + M_{1L}^{-1}(v - Aw), \end{aligned} \tag{5.28}$$

which corresponds to

$$M_{2L-H2}^{-1} = M_{1L}^{-1} + \left(I - M_{1L}^{-1} A\right) M_C^{-1}. \tag{5.29}$$

A symmetric two-level hybrid preconditioner can be obtained by applying M_{1L}^{-1} before and after the coarse-level correction, provided that A, M_{1L} and M_C are symmetric. sweeping back through the δ-overlap partitions of W.

5.2.2 Basic parallel sparse linear algebra operators

In this section a description of the setup and the application of the two-level Schwarz preconditioners M_{2L-A}, M_{2L-H1} and M_{2L-H2} is given, in terms of basic sparse Linear Algebra operators, with the aim of designing parallel algorithms and software based on standard kernels.

Consider a data distribution in which the number m of subsets of the δ-overlap partition of W is equal to the number of available processors; processor i holds both the matrix rows with indices in W_i^0 and the corresponding components of the solution and right-hand side vectors. This assumption is consistent with common usage, e.g. employing a graph partitioning tool to distribute the data of a sparse linear system arising from the discretization of a PDE.

Consider, now, the classical AS preconditioner. Processor i contributes to the setup of M_{1L} by building the matrix A_i^δ through a suitable *extension*

of A_i^0 with the rows corresponding to $W_i^\delta \backslash W_i^0$, held by other processors. To apply M_{1L}^{-1}, i.e. to compute $M_{1L}^{-1}v$, with $v \in \Re^n$, the processor i first *gathers* the components of v with indices in $W_i^\delta \backslash W_i^0$, obtaining $v_i^\delta = R_i^\delta v$, then *solves locally* a linear system with matrix A_i^δ, obtaining $w_i^\delta = (A_i^\delta)^{-1} v_i^\delta$, and, finally, *updates* the components of w_i^δ with indices in $W_i^0 \cap W_j^\delta$, for $j \neq i$, to compute its own part of $\sum_i^m (R_i^\delta)^T w_i^\delta$ according to the initial data distribution. As explained in chapter 4.1, the RAS and the ASH preconditioners do not require the update and the gather operators, respectively.

The two-level Schwarz preconditioners require additional basic operators to setup M_C and to apply it in combination with M_{1L}, according to 5.25, 5.26 or 5.28. The construction of the coarse-space approximation A_C is strongly dependent on the choice of the restriction operator R_C; the same observation holds for computations of the type $M_C^{-1}v$. The combination of M_C and M_{1L} requires a *parallel vector sum* and, in the hybrid versions, a *parallel sparse matrix by vector multiplication*.

To define R_C, and hence A_C and M_C, the *smoothed aggregation* technique presented in [12, 77] is used. The basic idea is to obtain the coarse set of vertices W_C by grouping the vertices of W into disjoint subsets and to define the coarse-to-fine space transfer R_C^T by applying a suitable smoother to a simple prolongation operator, to improve the quality of the coarse-space correction.

Three main steps can be identified in the smoothed aggregation procedure: the *coarsening* of the vertex set W, to obtain W_C, the *construction of the prolongator* R_C^T, and its *application*, to obtain A_C. To perform the coarsening step, the aggregation algorithm sketched in algorithm 17 is used, which is a particular case of the one presented in [77]. It aggregates the vertices in W into disjoint subsets V_C^i, $i = 1, \ldots, n_C$, such that $\cup_{i=1}^{n_C} V_C^i = W$; the V_C^i's are assumed as coarse-space vertices. As observed in [12], this algorithm can generate aggregates with suitable "aspect ratios". However, due to its sequential nature, it is not suited for a straightforward implementation on a distributed-memory parallel computer. Parallel aggregation schemes have been developed, aimed at building a coarse vertex set that allows a good coarse-space correction, while limiting the cost of data communication (see [31] and the references therein). The simplest parallel algorithm is the so-called *decoupled aggregation*, where the processor i independently applies the sequential algorithm to the set of vertices W_i^0 assigned to it in the initial data distribution. This version is embarrassingly parallel, since it does not require any data communication. On the other hand, it can produce non-uniform aggregates near boundary vertices, i.e. near vertices adjacent to vertices in other processors, and is strongly dependent on the number of processors and on the initial partitioning of the matrix A. Nevertheless, this approach has

Algorithm 17 Aggregation Algorithm

Require: $U \leftarrow W$; $N \leftarrow 0$; $n_c \leftarrow 0$

! Identify the aggregates:
repeat
  ! Start a new aggregate
  pick $i \in U$; $V_C^i \leftarrow \{i\}$; $n_C \leftarrow n_C + 1$;
  $U \leftarrow U \backslash \{i\}$;
  ! Add all neighbors to the aggregate
  $T \leftarrow \{j \in U : a_{ij} \neq 0\}$;
  $V_C^i \leftarrow V_C^i \cup T$;
  $U \leftarrow U \backslash T$;
  ! Mark for further processing all the vertices
  ! adjacent to the current aggregate
  $T \leftarrow \{i \in U; \exists k \in V_C^i : a_{ik} \neq 0\}$;
  $N \leftarrow N \cup T$;
  $U \leftarrow U \backslash T$;
until (U = 0)
! Add the remaining points to existing aggregates
$U \leftarrow N$
while $(U \neq 0)$ **do**
  pick $i \in U$;
  find smallest V_C^j such that $a_{ik} \neq 0$ for some $k \in V_C^j$;
  $V_C^j \leftarrow V_C^j \cup \{i\}$;
  $U \leftarrow U \backslash \{i\}$;
end while

been shown to produce good results in practice [73], therefore it has been chosen it in our implementation.

The prolongator $P_C = R_C^T$ is built starting from a *tentative prolongator* $P \in \Re^{n \times n_C}$, defined as

$$P = (p_{ij}), \quad p_{ij} = \begin{cases} 1 & \text{if } i \in V_C^j \\ 0 & \text{otherwise} \end{cases} . \tag{5.30}$$

Note that P can be regarded as a piecewise constant interpolation operator from $\Re^{n_C}$ to $\Re^n$. P_C is obtained by applying to P a smoother $S \in \Re^{n \times n}$:

$$P_C = SP, \tag{5.31}$$

in order to remove oscillatory components from the range of the prolongator and hence to improve the convergence properties of the two-level Schwarz

method [12, 63]. A simple choice for S is the damped Jacobi smoother:

$$S = I - \omega D^{-1} A, \tag{5.32}$$

where the value of ω can be chosen using some estimate of the spectral radius of $D^{-1}A$ [12].

If the decoupled aggregation is used, the tentative prolongator P can be built locally by each processor. Indeed, the processor i can set the part of the matrix P that corresponds to its local aggregates. Taking into account that in each processor the non-zero entries of P are in the rows and in the columns corresponding to the local vertices and the local aggregates, respectively, P can be regarded as distributed by rows conformally to the matrix A. Hence, to build the prolongator $P_C = SP$, the processor i has to *gather* the remote rows of P with indices in $W_i^1 \backslash W_i^0$ and to compute a *sequential sparse matrix by sparse matrix multiplication*. A *diagonal scaling* of the local rows of A is also required to obtain S. Then, the distributed matrix $A_C = R_C A R_C^T = P_C^T A P_C$ can be computed by applying twice a *gather* plus a *sequential sparse matrix by sparse matrix* operator, to recover the remote rows of P_C and AP_C with indices in $W_i^1 \backslash W_i^0$ and to build the local rows of AP_C and $P_C^T(AP_C)$ in each processor. After this step an all-to-all communication can be used to replicate the coarse matrix on all processors, if such a replication is requested by the user.

According to 5.24, the coarse-space correction $w = M_C^{-1} v$ requires two *parallel sparse matrix by vector multiplications*, to obtain $z = R_C v$, and $w = R_C^T(A_C^{-1} z)$, where the computation of $A_C^{-1} z$ can be performed by using either direct or iterative methods. If A_C is replicated, an all-to-all communication is needed to have the whole vector v in each processor.

5.2.3 PSBLAS-based implementation of two-level Schwarz preconditioners

The software architecture of the two-level Schwarz preconditioners, as implemented inside the PSBLAS library, is depicted in figure 5.5. The main routines specifically developed for the preconditioners are shown in the upper part of the picture, while the parallel and sequential PSBLAS kernels used by them are shown in the lower part. The dashed-line and the solid-line boxes represent routines already available in PSBLAS 1.0 and new routines (now included in version 2.0), respectively.

The construction of the enlarged matrix A_i^δ, in the routine `psb_ovrbld`, is implemented by first building the A_i^δ descriptor and then gathering the matrix rows corresponding to the indices in $W_i^\delta \backslash W_i^0$. The latter operation has

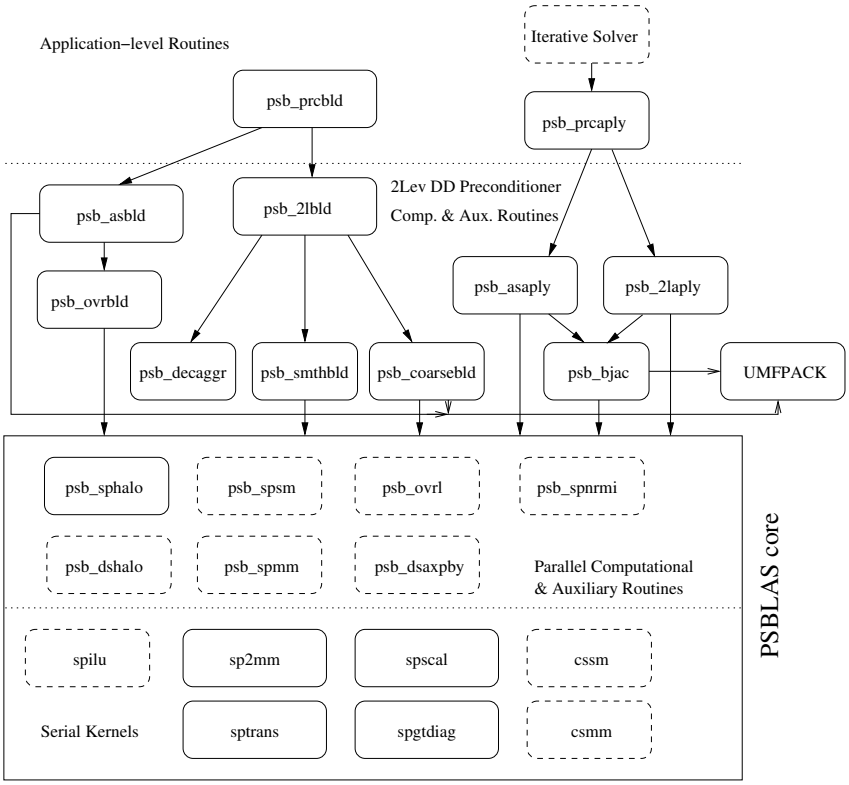

Figure 5.5: Software architecture of the package of PSBLAS-based two-level Schwarz preconditioners.

been implemented in the `psb_sphalo` routine; this routine can be regarded as a sparse version of the PSBLAS routine `psb_halo`, which acts on a dense matrix, distributed conformally to a sparse matrix, by recovering the rows that correspond to the overlap indices of the sparse matrix. Note that `psb_sphalo` has been put into the PSBLAS auxiliary routine set because it can be used in a more general context to build extended stencils, often required in PDE computations. As discussed in [13] and in chapter 4.1, the A_i^δ descriptor allows to use `psb_halo` and `psb_ovrl` to implement the restriction and prolongation operators R_i^δ and $(R_i^\delta)^T$.

The aggregation algorithm applied by the two-level preconditioner has been implemented in `psb_decaggr`. To build the smoothed prolongator P_C and the coarse matrix A_C, in `psb_smthbld` and `psb_coarsebld`, respectively, the set of PSBLAS serial kernels has been extended with modules performing the sparse matrix diagonal scaling and the sparse matrix by sparse matrix multiplication. The latter operation is performed via the integration into PSBLAS of the SMMP software [6], modified to take advantage of dynamic

memory management. It is worth noting that, although the sparse matrix multiplication functionality is not provided in the last SBLAS standard, the possibility of its future inclusion has been foreseen by the BLAS Technical Forum [29]. An estimate of the spectral radius of $D^{-1}A$ for the damping parameter ω, used in the definition of P_C is provided by computing the infinity norm of $D^{-1}A$ through the corresponding PSBLAS routine. The implementation of the restriction and prolongation operators in the coarse-space correction phase was obtained by using the PSBLAS sparse matrix by dense matrix multiplication module `psb_spmm`.

The solution of the linear systems with the matrices A_i^δ is performed using either an incomplete or a complete LU factorization. In the former case, this was accomplished by the PSBLAS serial kernels for ILU(0) and triangular system solve, `psb_spilu` and `psb_cssm`; in the latter, the UMFPACK package [23], version 4.4, was interfaced with the PSBLAS routines. The same options were made available for solving the systems with the replicated matrix A_C. To deal with the distributed A_C, also parallel block-Jacobi iterations has been implemented, with ILU(0) or LU on the diagonal block of A_C held by each processor.

Finally, all the previous functionalities were packaged at the application level into the setup `psb_prcbld` and application `psb_prcaply` routines.

5.2.4 Performance results

The Two-Level preconditioners implementation described in previous sections, has been tested by solving linear systems with coefficient matrices arising from simulations of the thermo-fluidynamics in an automotive engine and of the thermal diffusion in some solids.

The automotive engine test cases are snapshots from the simulation of a direct injection diesel engine, from a commercial automotive manufacturer. The matrices arise from the discretization of the pressure correction equation in the implicit phase of a semi-implicit algorithm (ICED-ALE [47]) for the solution of unsteady Navier-Stokes equations for compressible flows, as implemented in the KIVA application software modified to make use of the PSBLAS linear solvers [36]. The discretization mesh contains approximately 100000 control volumes; during the simulation the size of the actual domain varies, because mesh layers are activated/deactivated following the piston movement. The matrices correspond to different positions of the piston inside the engine cylinder; they have a symmetric sparsity pattern, with no more than 19 non-zero entries per row. The results concerning two matrices, named *kivap1* and *kivap2*, with dimensions 86304 and 42204, respectively are reported here.

Two other matrices have been considered, that arise from the discretization

of the steady-state thermal diffusion equation in solids. The first one, named *therm2D*, is of dimension 600000 and comes from a 2D model of a homogeneous plate, with a central finite-difference discretization scheme on a regular mesh. The second matrix, *therm3D*, has about 1 million rows and has been extracted from an experimental finite volume code that deals with the steady thermal conduction in materials with variable conductivity. A central discretization scheme is used in the code, including the deferred-correction approach proposed in [34] for handling non-orthogonal computational meshes. The simulation considered here concerns an aluminium Diesel engine piston discretized by using a tetrahedral mesh, with prescribed heat flux on the piston head and prescribed temperatures on the remaining surfaces.

The tests discussed here have been carried out on a cluster with dual-processor nodes, installed at the Innovative Computing Laboratory of the University of Tennessee at Knoxville. Each node has an AMD Opteron dual-processor (model 240, 1.4 GHz), running the Debian Linux 3.1 operating system with kernel 2.6.13, and 2 GBytes of memory; the nodes are connected with Myrinet network interfaces. The tests have been run on 32 nodes, i.e. on 64 processors. A development snapshot of the GNU compilers version 4.2, including both the C and Fortran 95 compilers, was used in conjunction with the specific MPI implementation for the Myrinet interface.

Table 5.1 shows that Additive Two-Level Schwarz preconditioning does not provide good results when compared to Multiplicative Two-Level Schwarz preconditioning in terms of number of iterations having a comparable cost measured in time/iteration. For this reason and sake of space, only timing and speedup results for RAS, with overlap 0 (Block-Jacobi) and 1, and for two variants of the two-level hybrid preconditioner 2LH-post, using RAS with overlap 1 at the fine level are presented. The coarse-space matrix is distributed among the processors and four Block-Jacobi sweeps are applied to the corresponding system; the first variant uses the ILU factorization on the diagonal blocks, while the second uses the LU factorization implemented in UMFPACK. In all the preconditioners, the systems arising in the application of RAS are solved by ILU. The preconditioners are applied as right preconditioners with the BiCGSTAB solver available in PSBLAS, choosing the null vector as starting guess. The iterations are stopped when the ratio between the 2-norms of the residual and of the right-hand-side is less than 10^{-6}; a maximum number of 1000 iterations is also set, but it is never reached in the tests. A row-block distribution of the matrices is used, where each processor holds (approximately) equal-sized blocks of consecutive rows, according to the well-known BLACS one-dimensional pure-block mapping. A conformal distribution is applied to the right-hand side and solution vectors. This choice implicitly defines a domain decomposition such that the number of subdomains is equal

to the number of processors. Results of a comparison of the PSBLAS package with well-established software implementing Schwarz preconditioners, carried out on the engine simulation and the 2D thermal diffusion matrices using other Linux clusters, are reported in [13].

therm2D								
	2LH-add, Block Jacobi with ILU				2LH-mult-post, Block Jacobi with ILU			
np	It.	Setup	Solve	Total	It.	Setup	Solve	Total
1	245	3.4200	112.9000	116.4000	183	3.1180	95.8900	99.0100
2	246	2.0850	89.6600	91.7500	182	1.8420	59.2900	61.1300
4	231	1.1280	48.1000	49.2300	190	0.9694	30.6400	31.6100
8	232	0.5205	17.4900	18.0100	184	0.4858	15.0500	15.5300
16	227	0.2946	7.9110	8.2060	171	0.2641	6.8410	7.1050
32	264	0.1734	4.7420	4.9160	175	0.1621	3.4830	3.6460
64	257	0.1207	3.0110	3.1320	176	0.1165	2.4460	2.5620
kivap1								
	2LH-add, Block Jacobi with ILU				2LH-mult-post, Block Jacobi with ILU			
np	It.	Setup	Solve	Total	It.	Setup	Solve	Total
1	10	1.5180	1.2400	2.7580	6	1.5660	1.0760	2.6420
2	10	0.9798	0.9476	1.9270	6	0.8857	0.5305	1.4160
4	11	0.5115	0.5374	1.0490	6	0.4909	0.2881	0.7790
8	12	0.3039	0.2941	0.5980	7	0.2924	0.1876	0.4800
16	12	0.1805	0.1612	0.3417	7	0.1837	0.1129	0.2966
32	12	0.1365	0.1124	0.2489	7	0.1372	0.0814	0.2186
64	12	0.1101	0.0873	0.1974	7	0.1194	0.0626	0.1820

Table 5.1: Comparison between Additive and Multiplicative Two-Level Schwarz preconditioning.

For all the matrices Tables 5.2-5.5 show the number of BiCGSTAB iterations (It), the execution times for the preconditioner setup (Setup) and for the solution of the preconditioned system (Solve), and the total times (Total), for NP = 1, 2, 4, 8, 16, 32, 64 processors. All the times are measured in seconds and are mean values over six executions. With therm3D it has not been possible to run the version of 2LH-post using the LU factorization for NP = 1, 2, because of the high memory requirements.

As expected, for kivap1 and kivap2 the best execution times are obtained with the RAS preconditioner, since the coarse-level correction is known to have a mild impact on matrices that do not come for pure elliptic problems.

kivap1								
	RAS, overlap 0				RAS, overlap 1			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	12	.2645	1.0830	1.3475	12	0.2658	1.1260	1.3918
2	16	0.1296	0.7101	0.8396	13	0.1613	0.6258	0.7871
4	16	0.0636	0.3706	0.4342	13	0.1170	0.3474	0.4643
8	20	0.0311	0.2276	0.2587	14	0.0789	0.2046	0.2834
16	22	0.0157	0.1330	0.1488	15	0.0579	0.1281	0.1860
32	23	0.0081	0.0777	0.0858	14	0.0528	0.0828	0.1356
64	24	0.0045	0.0388	0.0433	14	0.0446	0.0610	0.1056
	2LH-post, Block-Jacobi with ILU				2LH-post, Block-Jacobi with LU			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	6	1.5660	1.0760	2.6420	7	4.4300	1.5160	5.9470
2	6	0.8857	0.5305	1.4160	7	1.3360	0.8590	2.1950
4	6	0.4909	0.2881	0.7790	6	0.5737	0.3558	0.9295
8	7	0.2924	0.1876	0.4800	6	0.3207	0.1761	0.4968
16	7	0.1837	0.1129	0.2966	6	0.1915	0.0990	0.2905
32	7	0.1372	0.0814	0.2186	7	0.1412	0.0804	0.2215
64	7	0.1194	0.0626	0.1820	7	0.1213	0.0613	0.1826

Table 5.2: Iteration numbers and execution times, in seconds, for kivap1.

kivap2								
	RAS, overlap 0				RAS, overlap 1			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	38	0.1250	1.6560	1.7810	38	0.1251	1.7060	1.8310
2	55	0.0610	1.1490	1.2100	44	0.0762	1.0030	1.0790
4	55	0.0292	0.5909	0.6201	45	0.0639	0.5913	0.6552
8	73	0.0145	0.3982	0.4127	66	0.0394	0.4719	0.5114
16	87	0.0071	0.2476	0.2547	65	0.0294	0.2768	0.3061
32	95	0.0034	0.1104	0.1139	77	0.0253	0.2060	0.2313
64	126	0.0018	0.1116	0.1134	95	0.0216	0.1550	0.1766
	2LH-post, Block-Jacobi with ILU				2LH-post, Block-Jacobi with LU			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	19	0.7097	1.4400	2.1500	12	1.0910	1.0290	2.1210
2	20	0.3922	0.8013	1.1940	16	0.4615	0.8029	1.2640
4	20	0.2460	0.4728	0.7188	16	0.2623	0.3973	0.6596
8	25	0.1364	0.3271	0.4634	19	0.1415	0.2542	0.3958
16	26	0.0875	0.2142	0.3017	24	0.0908	0.1950	0.2858
32	29	0.0669	0.1649	0.2318	29	0.0693	0.1664	0.2357
64	37	0.0550	0.1713	0.2263	40	0.0552	0.1864	0.2416

Table 5.3: Iteration numbers and execution times, in seconds, for kivap2.

For the kivap matrices both the two-level preconditioners lead to a reduction of the iterations with respect to RAS, but this reduction is not strong enough to balance the larger execution times needed for building and applying the coarse-space corrections. The two-level preconditioners are more effective on therm2D and therm3D, that arise from the discretization of elliptic PDEs. However, while for therm2D the smallest execution times are obtained by using 2LH-post with the LU factorization, on therm3D the two-level preconditioners

therm2D								
	RAS, overlap 0				RAS, overlap 1			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	614	0.2193	172.9000	173.1000	614	0.2212	187.5000	187.7000
2	749	0.1183	108.5000	108.7000	834	0.1229	130.0000	130.1000
4	775	0.0621	54.7900	54.8500	970	0.0682	75.0900	75.1600
8	751	0.0341	26.9400	26.9700	741	0.0410	29.4700	29.5100
16	884	0.0214	15.4100	15.4300	783	0.0285	15.2000	15.2300
32	778	0.0141	6.7300	6.7440	680	0.0220	6.7170	6.7390
64	906	0.0113	5.0490	5.0600	812	0.0225	5.6390	5.6610
	2LH-post, Block-Jacobi with ILU				2LH-post, Block-Jacobi with LU			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	183	3.1180	95.8900	99.0100	5	10.2500	3.8210	14.0700
2	182	1.8420	59.2900	61.1300	18	4.0710	12.8300	16.9000
4	190	0.9694	30.6400	31.6100	26	1.8180	8.6890	10.5100
8	184	0.4858	15.0500	15.5300	34	0.7927	5.3530	6.1460
16	171	0.2641	6.8410	7.1050	57	0.3766	3.9230	4.2990
32	175	0.1621	3.4830	3.6460	76	0.2042	2.1980	2.4020
64	176	0.1165	2.4460	2.5620	106	0.1343	1.6330	1.7680

Table 5.4: Iteration numbers and execution times, in seconds, for therm2D.

therm3D								
	RAS, overlap 0				RAS, overlap 1			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	56	0.4348	27.2500	27.6800	56	0.4259	29.8300	30.2500
2	63	0.2237	15.7600	15.9900	55	0.2484	15.0400	15.2900
4	57	0.1270	10.3400	10.4700	61	0.1770	12.2400	12.4200
8	66	0.0773	6.3240	6.4010	57	0.1327	6.2150	6.3470
16	63	0.0383	2.1370	2.1760	58	0.1014	2.4040	2.5050
32	70	0.0247	1.2200	1.2440	59	0.0895	1.3940	1.4830
64	95	0.0185	0.9156	0.9341	71	0.0874	1.0920	1.1800
	2LH-post, Block-Jacobi with ILU				2LH-post, Block-Jacobi with LU			
NP	It	Setup	Solve	Total	It	Setup	Solve	Total
1	16	8.1740	16.0600	24.2300	–	——	——	——
2	16	4.9520	11.1000	16.0500	–	——	——	——
4	16	2.9370	7.8240	10.7600	7	158.8000	22.9800	181.8000
8	16	1.7560	4.3310	6.0870	8	44.1400	10.8800	55.0200
16	16	0.8118	1.8100	2.6220	10	10.2800	4.9550	15.2400
32	16	0.5323	1.0990	1.6310	13	2.4140	2.3730	4.7870
64	19	0.3962	0.8329	1.2290	17	0.8027	1.3300	2.1330

Table 5.5: Iteration numbers and execution times, in seconds, for therm3D.

are outperformed by RAS, except in a few cases. A closer look at the execution times of 2LH-post with LU shows that this preconditioner requires a much greater setup time than the RAS variants, because of the very large size of the matrix and of the sparsity structure, arising from a nine-point discretization stencil; conversely, the solution time is smaller for the two-level preconditioner.

This is interesting in real applications for two reasons. First, it is possible

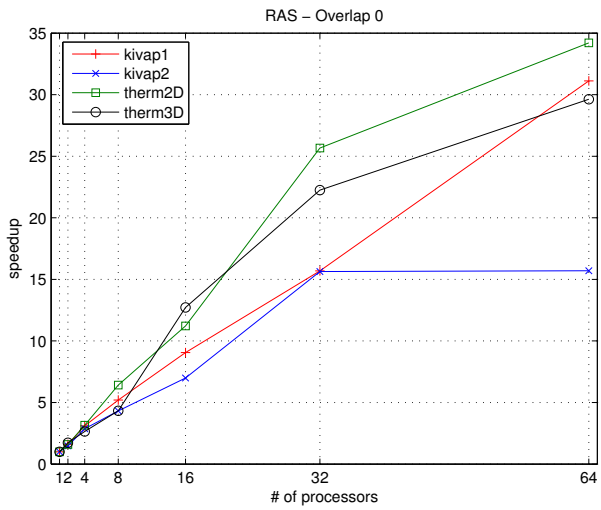

Figure 5.6: Speedups for RAS with overlap 0.

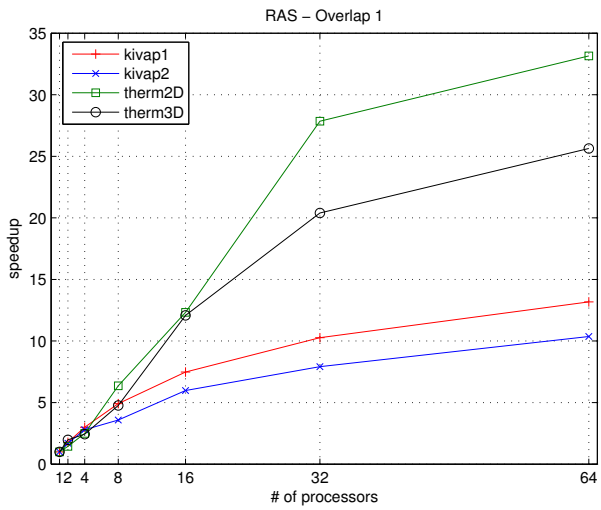

Figure 5.7: Speedups for RAS with overlap 1.

to reuse some of the aggregation and coarse matrix data structures in all the cases where the aggregation is purely topological and the matrix pattern has not changed between two successive invocations of the solver. Moreover, it may be feasible to reuse the same preconditioner for a number of outer iterations of a nonlinear solver if the coefficient matrix does not change too

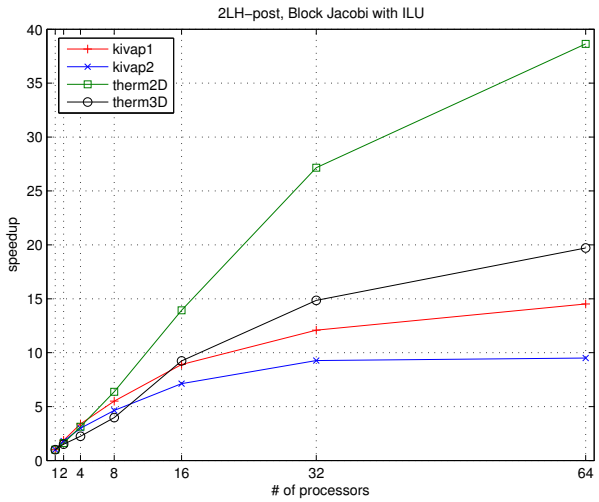

Figure 5.8: Speedups for 2LH-post with four Block-Jacobi sweeps and ILU factorization of the blocks.

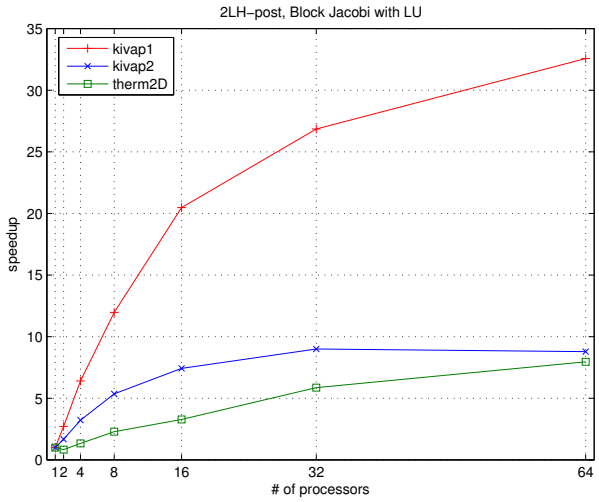

Figure 5.9: Speedups for 2LH-post with four Block-Jacobi sweeps and LU factorization of the blocks.

much. In Figures 5.6-5.9 the speedups of the four selected preconditioners are plotted for each of the test matrices; the values concerning 2LH-post with the LU factorization on therm3D are missing, since running this test for NP = 1, 2 is not possible on the system used for this measures. The speedups

have been computed using the total times, including the time needed to build the preconditioner. The overall results are affected by the increase in the number of BiCGSTAB iterations that can be observed with the increase in the number of processors.

As expected, RAS with overlap 0 generally shows good speedup values. The only exceptions are provided by kivap2 on 32 and 64 processors; in this case the speedup tends to saturate because of the relatively small size of the local portion matrix. The highest speedup value on 64 processor is close to 35 and is obtained with therm2D. By using RAS with overlap 1, a speedup decrease can be observed on the engine simulation and the 3D thermal diffusion matrices; speedup values comparable with those corresponding to overlap 0 are obtained on therm2D, which has a sparsity pattern coming from a simple discretization stencil. For kivap1 and kivap2 the speedup of 2LH-post with ILU has a very close behavior to that of RAS with overlap 1. On therm2D, 2LH-post with ILU shows a small speedup increase with respect to RAS, with a value close to 40 on 64 processors. On therm3D, instead, the speedup for 2LH-post with ILU is lower than for the RAS cases; this is mainly due to the size and the sparsity structure of the matrix. The situation is different for 2LH-post with LU. The highest speedup is now achieved on kivap1, for which the number of BiCGSTAB iterations is approximately constant, but the time required by the LU factorization significantly reduces in going from 1 to 64 processors. On kivap2 the speedup behavior is comparable with that of the other two-level preconditioner, while on therm2D a strong speedup reduction is observed, which is mainly due to the large increase of the iterations, and hence of the solve time, as the number of processors grows.

Part IV

Applications of the PSBLAS library

Chapter 6

PSBLAS applications

Contents

6.1 Overview

As already discussed in chapter 2, the PSBLAS library has been designed to address the solution of sparse systems arising from the discretization of PDEs through iterative solvers. In this context it is very important to handle efficiently the movement of data among subdomains defined by a graph partitioning method (as discussed in chapter 2) while providing flexibility and ease of usage to the user. This makes the PSBLAS library suitable also for applications using different PDE discretization and solution techniques.

This chapter presents a brief description of application which arise from different contexts or which are based on different approaches than the solution of discretized PDEs with iterative solvers. Some of these applications are still work-in-progrees and only preliminary results are available; moreover the related documentation is still in beta release stage. Anyway the aim of this discussion is just to give the reader an idea of the range of applications of the PSBLAS software package.

Computational Fluid Dynamics and the KIVA application The development of the PSBLAS software package has been mainly driven by the issues arising from applications which are based on the discretization of Partial Differential Equations. In this sense, Computational Fluid Dynamics (CFD) studies that support automotive engine design represent a challenging testbed and an abundant source of suggestions for the development and enhancement of the PSBLAS library.

The use of CFD, indeed, have revealed to be of great support for both the design of automotive engines and the experimental work in order to quickly achieve the projects targets while reducing the product development costs. However, considerable work is still needed in the developemnt of suitable software tools since CFD simulation of realistic industrial applications may take many hours or even weeks that not always agrees with the very short development times that are required to a new project in order to be competitive in the market.

The KIVA code [56] solves the complete system of general time-dependent Navier-Stokes equations and it is probably the most widely used code for internal combustion engines modeling. Its success mainly depends on its open source nature. KIVA has been significantly modified and improved by researchers worldwide, especially in the development of sub-models to simulate the important physical processes that occurs in an internal combustion engine (i.e. fuel-air mixture preparation and combustion).

The next section (section 6.2) describes in details how the KIVA code can benefit from the usage of the PSBLAS features. In particular it shows how execution time can be reduced thanks to parallelism provided by PSBLAS and thanks to the usage of more suitable solvers which also improve the accuracy of the computations [8].

Lattice-Boltzmann Methods LBM takes inspiration from the idea of solving fluid flows through a microscopic kinetic approach, trying to mathematically describe movements and interactions of the small particles that constitute the flow with the assumption that the solute concentrations are sufficiently low not to influence the solvent flow.

At the turn of the 1980s, the Lattice Boltzmann Method (LBM) has been proposed as an alternative approach to solve fluid dynamics problems [10, 70] and due to the refinements and the extensions of the last years [9, 41, 53], it has been used to successfully compute a number of nontrivial fluid dynamics problems, from incompressible turbulence to multiphase flow and bubble flow simulations. The main advantages of LBM with

respect to conventional CFD are its simpler mechanism for doing dynamics, its easy numerical implementation and its intrinsic parallelism.

The most severe limitation of the original LB method is the uniform Cartesian grid on which the LBM must be constructed, requiring the approximation of a curved solid boundary by a series of stair steps. This represents a particularly severe limitation for practical engineering purposes especially when there is a need for high resolutions near the body or the walls. Among the recent advances in lattice Boltzmann research that have lead to substantial enhancement of the capabilities of the method to handle complex geometries [9, 41, 53], a particularly remarkable option is to use irregular lattices by changing the solution procedure from the original "stream and collide" to a finite volume technique [59, 75, 76].

The basic differential form of the Lattice Boltzmann equation can be formulated in a finite-volume framework (see [76]) which can in turn be implemented with a series of sparse matrix-vector operations. Moreover, in most applications of LB, it is necessary to employ large discretization meshes; thus, it is appropriate to use parallel computing techniques. All of these issues make PSBLAS a suitable choice for the implementation of applications based on the Lattice Boltzmann which doesn't involve the solution of sparse linear systems as discussed in [38].

Thermal Diffusion The PSBLAS solvers have been used to develop an application called Vulcan (still work-in progress) which is is a finite–volume application for the solution of the Fourier's equation (6.1) for the thermal diffusion in solid bodies.

$$\rho c \frac{\partial T}{\partial t} = \mathrm{div}\,(\lambda \mathrm{grad} T) + q \tag{6.1}$$

In particular, the steady conduction in materials with variable thermal conductivity λ has been investigated, in order to assess the validity of the iterative procedure for dealing with the non–linear effects induced by the non–constant diffusion coefficient.

The discretization scheme adopted in the code is the central difference, with an extra source–term for handling the non–orthogonality of the computational mesh, based on the deferred correction approach proposed by Ferziger and Peric [34].

Discretizing the equation the equation (6.1) yields a linear system $Ax = b$ where A is a matrix whose simmetry reflects the ellipticity of the problem.

Three kinds of physical boundary conditions are implemented: prescribed temperature, adiabatic wall and prescribed heat flux. From a numerical point of view, the first one is a Dirichlet boundary condition, while the second and third are Neumann ones. The implementation consists in the introduction of special source terms which add proper contributions to the right hand side b and to the main–diagonal coefficient of each row of A. The term q in the (6.1) represents a heat source which can be uniformly distributed or not.

The loop of external iterations needed for dealing with the semi–implicit correction of the non–orthogonality and with the non–linearity of the diffusion coefficient implies the use of PSBLAS matrix rigeneration (see section 2.5) which improves the speed of the assembling phase in re–iterated solving process.

The overall procedure has showed a second order accuracy when compared to analytical solutions existing in literature.

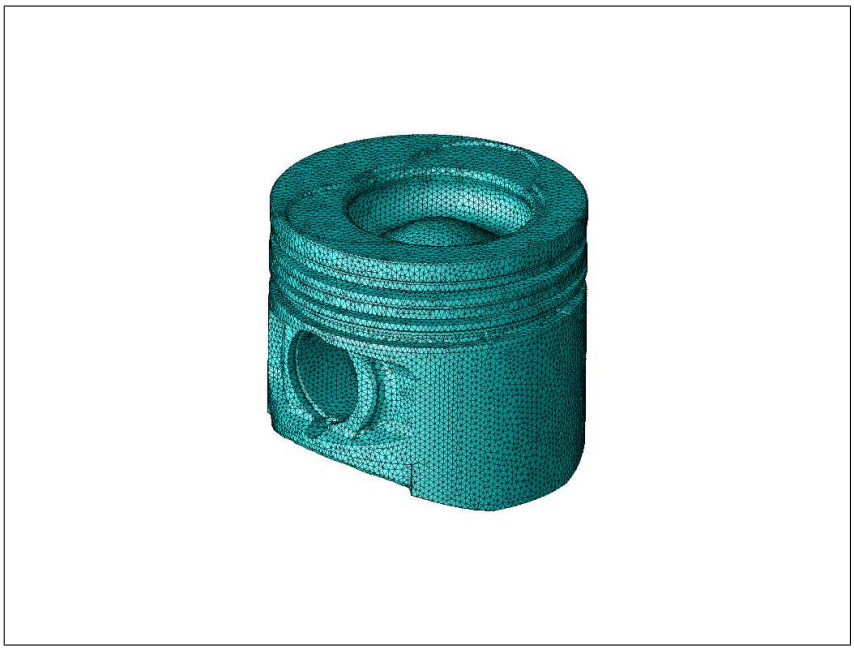

Figure 6.1: Diesel engine piston: mesh

The code has been tested with the tetrahedral mesh of a Diesel engine piston depicted in figure 6.1, choosing aluminum as material and applying a prescribed heat flux on the piston head and prescribed temperatures on the remaining surfaces.

6.2 Embedding PSBLAS solvers in the KIVA application

The following sections will review the basic mathematical model of the Navier-Stokes equations as discretized in the application presented in [8]; the approach to the linear system solution based on the library routines from [35, 39] will be also be described.

6.2.1 Mathematical model

The mathematical model of KIVA-3 is the complete system of general *unsteady Navier-Stokes equations*, coupled with chemical kinetic and spray droplet dynamic models. In the following the equations for fluid motion are reported.

- Species continuity:

$$\frac{\partial \rho_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}) = \nabla \cdot [\rho D \nabla (\frac{\rho_m}{\rho})] + \dot{\rho}_m^c + \dot{\rho}_m^s \delta_{ml} \tag{6.2}$$

  where ρ_m is the mass density of species m, ρ is the total mass density, $\mathbf{u}$ is the fluid velocity, $\dot{\rho}_m^c$ is the source term due to chemistry, $\dot{\rho}_m^s$ is the source term due to spray and δ is the Dirac delta function.

- Total mass conservation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = \dot{\rho}^s \tag{6.3}$$

- Momentum conservation:

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\, \mathbf{u}) = -\frac{1}{\alpha^2} \nabla p - A_0 \nabla (\frac{2}{3} \rho k) + \nabla \cdot \overline{\sigma} + \mathbf{F}^s + \rho \mathbf{g} \tag{6.4}$$

  where $\overline{\sigma}$ is the viscous stress tensor, $\mathbf{F}^s$ is the rate of momentum gain per unit volume due to spray and $\mathbf{g}$ is the constant specific body force. The quantity A_0 is zero in laminar calculations and unity when turbulence is considered.

- Internal energy conservation:

$$\frac{\partial (\rho I)}{\partial t} + \nabla \cdot (\rho I \mathbf{u}) = -p \nabla \cdot \mathbf{u} + (1 - A_0) \overline{\sigma} : \nabla \mathbf{u} - \nabla \cdot \mathbf{J} + A_0 \rho \epsilon + \dot{Q}^c + \dot{Q}^s \tag{6.5}$$

  where I is the specific internal energy, the symbol : indicates the matrix product, $\mathbf{J}$ is the heat flux vector, $\dot{Q}^c$ and $\dot{Q}^s$ are the source terms due to chemical heat release and spray interactions.

Furthermore the standard $K - \epsilon$ equations for the turbulence are considered, including terms due to interaction with spray.

Numerical method

The numerical method employed in KIVA-3 is based on a variable step *implicit Euler* temporal finite difference scheme, where the time steps are chosen using accuracy criteria. Each time step defines a cycle divided in three phases, corresponding to a physical splitting approach. In the first phase, spray dynamic and chemical kinetic equations are solved, providing most of the source terms; the other two phases are devoted to the solution of fluid motion equations [57]. The spatial discretization of the equations is based on a *finite volume method*, called the *Arbitrary Lagrangian-Eulerian method* [62], using a mesh in which positions of the vertices of the cells may be arbitrarily specified functions of time. This approach allows a mixed Lagrangian-Eulerian flow description. In the Lagrangian phase, the vertices of the cells move with the fluid velocity and there is no convection across cell boundaries; the diffusion terms and the terms associated with pressure wave propagation are implicitly solved by a modified version of the SIMPLE (Semi Implicit Method for Pressure-Linked Equations) algorithm [58].Upon convergence on pressure values, implicit solution of the diffusion terms in the turbulence equations is approached. Finally, explicits methods, using integral sub-multiple time-steps of the main computational time step, are applied to solve the convective flow in the Eulerian phase.

6.2.2 Algorithmic issues

One of the main objectives of the work on the KIVA code described in [35] was to show that general purpose solvers, based on up-to-date numerical methods and developed by experts, can be used in specific application codes, improving the quality of numerical results and the flexibility of the codes as well as their efficiency.

The original KIVA code employs the Conjugate Residual method, one member of the Krylov subspace projection family of methods [7, 44, 49, 64]. Krylov subspace methods originate from the Conjugate Gradient algorithm published in 1952, but they became widely used only in the early 80s. Since then this field has witnessed many advances, and many new methods have been developed especially for non-symmetric linear systems. The rate of convergence of any given iterative method depends critically on the eigenvalue spectrum of the linear system coefficient matrix. To improve the rate of convergence it is often necessary to *precondition* it, i.e. to transform the system into an equivalent one having better spectral properties. What follows describes a work started with the idea of introducing new linear system solvers and more sophisticated preconditioners in the KIVA code; a number

of issues related to the code design and implementation had to be tackled to achieve significant results.

Code design issues

KIVA-3 is a finite-volume code in which the simulation domain is partitioned into hexahedral cells, and the differential equations are integrated over the cell to obtain the discretized equation, by assuming that the relevant field quantities are constant over the volume. The scalar quantities (such as temperature, pressure and turbulence parameters), are evaluated at the centers of the cells, whereas the velocity is evaluated at the vertices of the cells. The cells are represented through the coordinates of their vertices; the vertex connectivity is stored explicitly in a set of three connectivity arrays, from which it is possible by repeated lookup to identify all neighbours, as shown in figure 6.2. The original implementation of the CR solver for linear employs

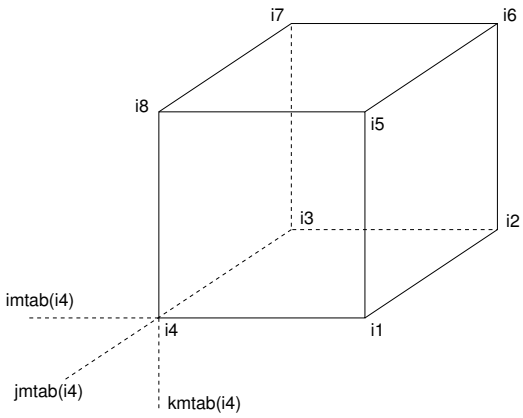

Figure 6.2: Vertex numbering for a generic control volume

a *matrix-free* approach, i.e. the coefficient matrix is not formed explicitly, but its action is computed in an equivalent way whenever needed; this is done by applying the same physical considerations that would be needed in computing the coefficients: there is a main loop over all cells, and for each cell the code computes the contribution from the given cell into the components corresponding to all adjacent cells (including itself), from $i1$ through $i8$. This is a "scatter" approach, quite different from the usual "gather" way of computing a matrix-vector product.

The major advantage of a matrix-free implementation is in terms of memory occupancy. However it constraints the kind of preconditioners that can be applied to the linear system; in particular, it is difficult to apply

preconditioners based on incomplete factorizations. Moreover, from an implementation point of view, data structures strictly based on the modeling aspects of the code do not lend themselves readily to transformations aimed at achieving good performance levels on different architectures.

The above preliminary analysis has influenced the design of the interface to the sparse linear solvers and support routines in [39] with the following characteristics:

1. The solver routines are well separated into different phases: matrix generation, matrix assembly, preconditioner computation and actual system solution;

2. The matrix generation phase requires an user supplied routine that generates (pieces of) the rows of the matrix in the global numbering scheme, according to a simple storage scheme, i.e. coordinate format;

3. The data structures used in the solvers are parametric, and well separated from those used in the rest of the application.

6.2.3 Integration of the numerical library

The basic groundwork for the parallelization and integration of the numerical library has been laid out at the time of [35], which is going to be briefly reviewed below.

The code to build the matrices coefficients has been developed starting from the original solver code: the solvers in the original KIVA code are built around routines that compute the residual $r = b - Ax$, and the right hand side b and the matrix A have been built starting from these. Since the solution of equations for thermodynamic quantities (such as temperature, pressure and turbulence) requires cell center and cell face values, the non-symmetric linear systems arising from temperature, pressure and turbulence equations have coefficient matrices with the same symmetric sparsity pattern, having no more than 19 nonzero entries per row. In the case of the linear systems arising from the velocity equation, following the vectorial solution approach used in the original code, the unknowns are ordered first with respect to the three Cartesian components and then with respect to the grid points. The discretization scheme leads to non-symmetric coefficient matrices with no more than 27 entries per (block) row, where each entry is a 3×3 block.

Algorithmic Improvements

The original KIVA-3 code solution method, the Conjugate Residual method, is derived under the hypothesis of a symmetric coefficient matrix; thus, there

is no guarantee that the method should converge on non-symmetric matrices such as the ones encountered in KIVA. Therefore alternative solution and preconditioning methods has been tested with the aim to achieve good performance and reliability. Since the convergence properties of any given iterative method depend on the eigenvalue spectrum of the coefficient matrices arising in the problem domain, there no reason to expect that a single method should perform optimally under all circumstances [55, 64, 72]. Thus an experimental approach has been adopted, in searching for the best compromise between preconditioning and solution methods. The Bi-CGSTAB method has been choosen for all of the linear systems in the SIMPLE loop; the critical solver is that for the pressure correction equation, where a block ILU preconditioner has been used, i.e. an incomplete factorization based on the local part of A. The BiCGSTAB method always converged, usually in less than 10 iterations, and practically never in more than 30 iterations, whereas the original solver quite often would not converge at all.

The new code also gains in performance for two other reasons:

1. The termination of the SIMPLE loop is based on the amount of the pressure correction; thus a better solution for the pressure equation reduces the number of SIMPLE iterations needed.

2. The time-step chosen for the discretization of the non linear equations is no longer limited by the quality of the linear solvers.

Further research work on other preconditioning schemes is currently ongoing, and it is planned to include its results in future versions of the code [13, 22].

6.2.4 Parallelization issues and new developments

Since the time of [35] the code has undergone a major restructuring: FAST-EVP code is now based on the KIVA-3V version, and thus it is able to model valves. This new modeling feature has no direct impact on the SIMPLE solvers interface, but it is important in handling mesh movement changes. While working on the new KIVA-3V code base, he space allocation requirements have been reviewed, cleaning up a lot of duplications.

All computations in the code are parallelized with a domain decomposition strategy: the computational mesh is partitioned among the processors participating in the computation. This partitioning is induced by the underlying assumptions in the linear system solvers; however it is equally applicable to the rezoning phase. The support library routines allow to manage the necessary data exchanges throughout the code based on the same data structures employed

for the linear system solvers; thus, it is possible to define a unifying framework for all computational phases.

The rezoning phase is devoted to adjusting the grid points following the application of the fluid motion field; the algorithm is an explicit calculation that is based on the same "gather" and "scatter" stencils found in the matrix-vector products for the linear systems phase in figure 6.2. It is thus possible to implement in parallel the explicit algorithm by making use of the data movement operations defined in the support library [39].

The chemical reaction dynamics is embarassingly parallel, because it treats the chemical compounds of each cell independently.

For the spray dynamics model specific operators that follow the spray droplets in their movement have been developed, transferring the necessary information about the droplets whenever their simulated movement brings them across the domain partition boundaries.

Mesh movement

The simulation process modifies the finite volume mesh to follow the (imposed) piston and valve movement during the engine working cycle (see also Fig. 6.3. The computational mesh is first deformed by reassigning the positions of the finite volume surfaces in the direction of the piston movement, until a critical value for the cell aspect ratio is reached; at this point a layer is cut out (or added into) the mesh to keep the aspect ratio within reasonable limits. When this "snapper" event takes place it is necessary to repartition the mesh and to recompute the patterns of the linear system matrices. The algorithm for matrix assembly takes into account the above considerations by preserving the matrix structure between two consecutive "snapper" points, and recomputing only the values of the non-zero entries at each invocation of the linear system solver; this is essential to the overall performance, since the computation of the structure is expensive.

Similarly, the movement of valves is monitored and additional "snapper" events are generated accordingly to their opening or closing; the treatment is completely analogous to that for the piston movement.

6.3 Experimental results

The first major test case discussed is based on a high performance competition engine that was used to calibrate the software. The choice of this engine was due to the availability of measurements to compare against, so as to make sure not to introduce any modifications in the physical results. Moreover it

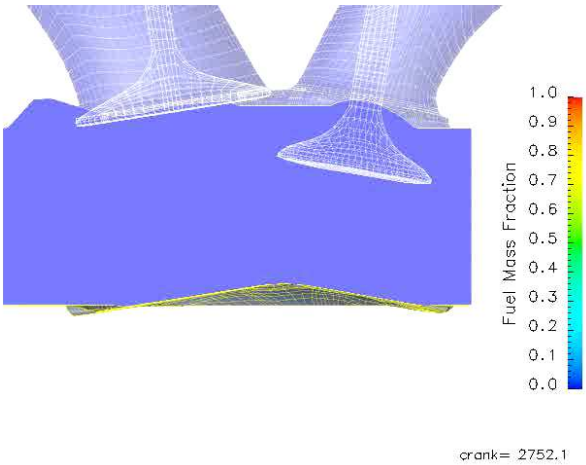

Figure 6.3: Competition engine simulation

Processes	Time steps	Total time (min)
1	2513	542
2	2515	314
4	2518	236
5	2515	186
6	2518	175
7	2518	149

Table 6.1: Competition engine timings

is a very demanding and somewhat extreme test case, because of the high rotation regime, high pressure injection conditions, and relatively small mesh size.

A section of the mesh, immediately prior to the injection phase, is shown in figure 6.3; the overall mesh is composed of approximately 200K control volumes. The simulated comprises 720 degrees of crank angle at 16000 rpm, and the overall timings are shown in table 6.1.

The computation has been carried out at NUMIDIA srl on a cluster based on Intel Xeon processors running at 3.0 GHz, equipped with Myrinet M3F-PCIXD-2 network connections. The physical results were confirmed to be in line with those obtained by the original code. Figure 6.4 shows the average value of the pressure in the combustion chamber. Figure 6.5 shows the partitioning of the mesh on 16 processors; the snapshot is taken close to the top dead center.

Another interesting test case is a complete test of a commercial engine

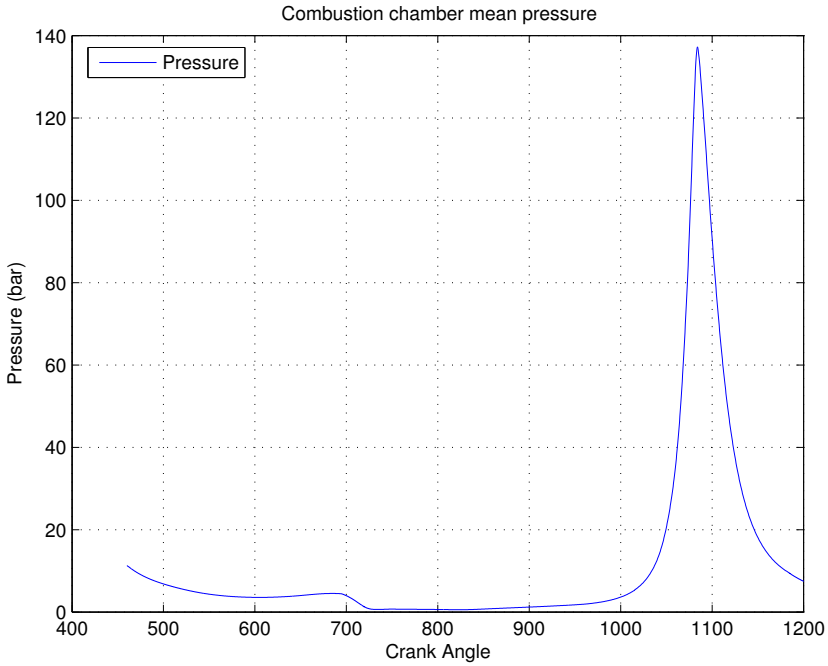

Figure 6.4: Competition engine average pressure

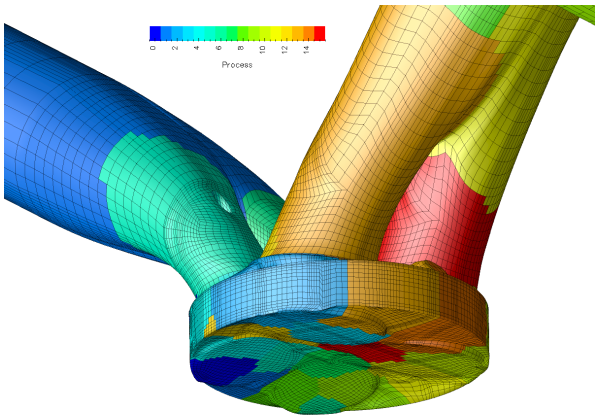

Figure 6.5: Mesh partitioning on 16 processes.

cylinder coupled with an air box, running at 8000 rpm; figure 6.6 shows the resulting airflow. The discretization mesh is composed of 483554 control

volumes; the simulation comprises the crank angle range from 188 to 720 degrees, with 4950 time steps, and it takes 703 minutes of computation on 9 nodes of the Xeon cluster. In this particular case the grid had never been tested on a serial machine, or on smaller cluster configurations, because of the excessive computational requirements; thus the parallelization strategy has enabled us to obtain results that would have been otherwise unreachable.

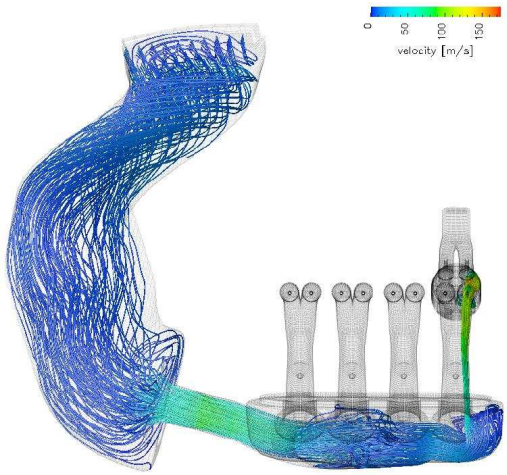

Figure 6.6: Commercial engine air flow results

Part V

Conclusions

Chapter 7

Conclusions

The topics discussed throughout the various sections of this thesis describe the activity carried out during a PhD course in Computer Science. All of the work presented has been developed in the context of Computational Science and Engineering. It has been already pointed out how important simulation has become either in scientific disciplines or in industrial processes. Many areas can substantially benefit from the usage of CSE tools to achieve a deeper understanding of phenomena that are either to expensive or too dangerous, if not impossible at all, to analyze by direct inspection. Examples are chemistry, biology, bioengineering, wheater forecast, automotive industry, electronic design automation, aircraft design...

The development of efficient and easy to use CSE tools is, thus, of key importance.

This thesis advocates how a multidisciplinary approach is almost essential to this aim. Based on this concept, Computational Sicence and Engineering can be defined as the overlap of three main areas (see [82]):

Science/Engineering this disciplines develop models that simulate natural phenomena;

Applied Mathematics this is the area where algorithms and numerical methods are developed for the solution of model coming from Science/Engineering;

Computer Science this area deals with the efficent implementation of numerical methods on computer architectures.

Each of these disciplines gives it contribution to the development and improvement of CSE tools and methods. Figure 7.1 (taken from [82]) shows how the development of new and more powerful numerical methods contributes to the efficiency of CSE tools (*top*) compared to the contribution offered by the advances in computer performances (*bottom*). In particular this figure

refers to methods for the solution of linear sparse systems some of which have been extensively discussed in chapters 1 and 5. Even if figure 7.1 is not

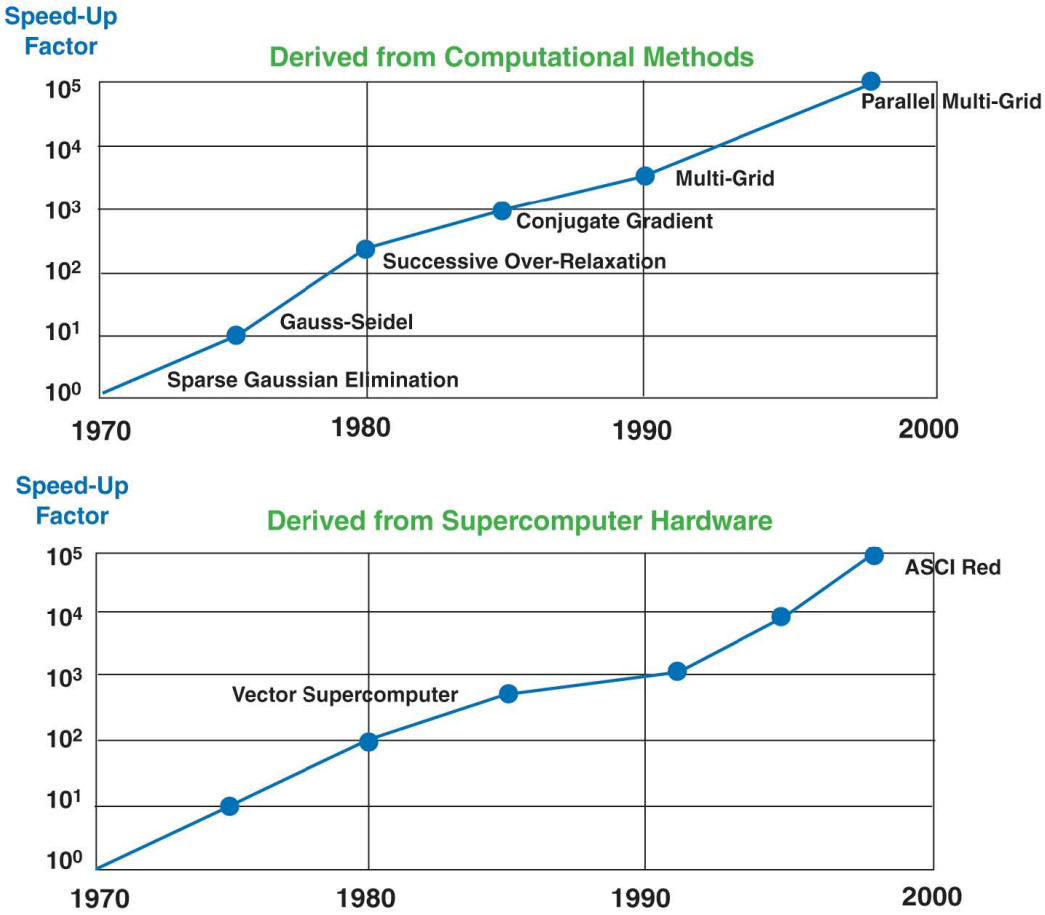

Figure 7.1: Comparison of the contributions of mathematical algorithms and computer hardware.

updated, the same trend is likely to be followed in the next years. This figure shows that as the complexity of numerical methods and computer hardware grows over the years, higher CSE skills have to be developed in order to provide efficient implementations of modern numerical methods that take full advantage of the most recent computer machinery.

To this extent, scientific and engineering applications represent an uncomparable source of inputs. A concrete knowledge of applied mathematics is essential to identify the numerical method that better suits the solution of a scientific problem. Finally substantial computer sience skills allow the development of

efficient implementations of numerical methods on modern and very complex computer architectures.

This thesis shows how it is possible to develop efficient yet flexible CSE tools (namely the PSBLAS library) based on knowledge of the three disciplines discussed above:

Computer Science chapter 3 presents an self-adaptive optimization technique for the matrix-vector product operation. This technique is based on the usage of a blocked matrix storage format (BCSR) where the dimension of the blocking provides different level of performance. The self-adaptativity consists on the automatic choice of the blocking dimension in such a way that execution times for the matrix-vector product are minimized. Results presented in section 3.3.3 show that this optimization technique, while providing extremely high portability, is capable of reducing the matrix-vector product execution time up to a factor of four (in the case of the Itanium2 architecture). However, as already discussed in section 3.3.3, the speedup provided by this optimization technique strongly depends on the architecture and inout data characteristics and, thus, it may happen that for some matrices on some architectures using the BCSR storage format doesn't provide any improvement. This is the case of most of the matrices in the testset on the MIPS and Xeon architectures.

Self-adaptativity is gaining a lot of ground in dense linear algebra. This is due to the fact that the performance of dense linear algebra code is only dependent on architecture characteristics. The ATLAS [81] project is a well known example of self-adaptive software package for the dense linear algebra. In sparse linear algebra, developing self-tunable source code is much more complex because the performance of each operation is strongly dependent also on the input data characteristics. This introduces the need for a run-time analisys phase whose cost may exceed the improvements gained.

However chapter 3 shows that self-adaptativity is worth being investigated even in sparse linear algebra.

Possible future directions include the integration of the proposed optimization technique into the PSBLAS library (which is, at the moment, work in progress). This implies that the BCSR optimizations must be extended to many ither operations different than the matrix-vector product. Moreover the same approach can be extended to other parametric storage formats for sparse matrices like, for example, the JAD variant implemented in the PSBLAS library (see section 3.1.3). Finally other

optimization techniques like cache blocking or TLB blocking may be exploited in a self-adaptive way.

Applied Mathematics Chapters 4 and 5 describe to preconditioning techniques for iterative linear system solvers. The Additive Schwraz and Two-Level Additive Schwarz are parallel preconditioners that are, respectively, based on Domain Decomposition and Multilevel methods described in sections 4.1 and 5.1. Both of them show very good numerical properties; the Additive Schwarz preconditioner may substantially reduce the number of iterations to convergence with respect to the Block Jacobi (BJA) preconditioner for most of the matrices that have been used for testing (i.e. those reported in section 4.4 and more). In the case of the Two-Level Additive Schwarz preconditioner, this reduction is even stronger: in the case of the "therm2D" matrix the number of iterations is reduced by more than an order of magnitude when the Two-Level preconditioner matrix is factorized with a complete factorization. Both of these preconditioners, however, have an expensive setup phase and also the application phase is considerably more expensive with respect to simple preconditioners as BJA. This means that in some cases the solver execution time may only barely reduced if even not increased. It is important to recall that the preconditioner setup cost may be substantially reduced considering that one preconditioner may be used more than once and that topological informations may be reused in subsequent preconditioners building.

Future work will be directed to the development of more general Multi-Level preconditioners: the Two-Level Additive Schwarz preconditioner shows very good numerical properties in the case where the Two-Level matrix is completely factorized; however a Two-Level preconditioning matrix may still to big to be completely factorized and, thus, the cost of the preconditioner setup and application phases too high. Higher level preconditioning matrices should be enough small to be easily factorize with a complete factorization.

Science/Engineering Section 6.2 shows how a general purpose CSE software (namely the PSBLAS library) can be easily used to build applications that solve models developed in Science or Engineering studies. The PSBLAS library has been used to replace the original solvers in the KIVA-III application for automotive engine design. The library integration has provided several benefilts to the application either from a numerical or a performance point of view. The PSBLAS library is being used also to develop other applications (see section 6) some of which (i.e. those

based on the Lattice-Boltzmann methods) are based on a completely different approach than the discretization of PDEs which is the main target of the PSBLAS development.

Future works include providing these applications more features that improve performance. Specifically, the possibility to reuse topological information in the preconditioner building phase during successive timestep is being investigated.

Bibliography

[1] METIS:unstructured graph partitioning and sparse matrix ordering system. http://www-users.cs.umn.edu/~karypis/metis/.

[2] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. also ENSEEIHT-IRIT Technical Report RT/APO/99/2.

[3] Mario Arioli, Iain Duff, and Daniel Ruiz. Stopping criteria for iterative solvers. *SIAM J. Matrix Anal. Appl.*, 13(1):138–144, 1992.

[4] O. Axelsson. On preconditioning and convergence acceleration in sparse matrix problems. (74-10), 1974.

[5] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.0, Argonne National Laboratory, August 2004.

[6] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (smmp).

[7] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk A. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.org/templates/Templates.html, 1994.

[8] G. Bella, A. Buttari, A. De Maio, F. Del Citto, S. Filippone, and F. Gasperini. Fast-evp: an engine simulation tool. In Springer, editor, *High Perfromance Computing and Communications. First International*

Conference, HPCC 2005, Proceedings, volume 3726 of *Lecture Notes in Computer Science*, pages 976–986, September 2005.

[9] G. Bella, S. Ubertini, and M. Bertolino. Computational fluid dynamics for low and moderate reynolds numbers through the lattice boltzmann method. *International Journal of Computational and Numerical Analysis and Applications*, 3:83–115, 2003.

[10] R. Benzi, S. Succi, and M. Vergassola. The lattice Boltzmann equation: theory and applications. *Phys. Rep.*, 222:145–197, December 1992.

[11] Achi Brandt. Algebraic multigrid theory: The symmetric case. *Appl. Math. Comput.*, 19(1-4):23–56, 1986.

[12] Marian Brezina and Petr Vanek. One black-box iterative solver. Technical Report UCD-CCM-106, 1, 1997.

[13] A. Buttari, P. D'Ambra, D. di Serafino, and S. Filippone. Extending psblas to build parallel schwarz preconditioners. In Springer, editor, *Applied Parallel Computing. State of the Art in Scientific Computing: 7th International Conference, PARA 2004, Lyngby, Denmark, June 20-23, 2004.*, volume 3732 of *Lecture Notes in Computer Science*, pages 593–602, February 2006.

[14] A. Buttari, P.D'Ambra, D. Di Serafino, and S. Filippone. 2levdd-psblas: a package of high-performance preconditioners for scientific and engineering applications. Technical Report RT-ICAR-NA-2005-20, Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR), 12 2005. submitted at "Applicable Algebra in Engineering, Communication and computing", Special Issue on "Computational Linear Algebra and Sparse Matrix Computations".

[15] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. Technical Report ICL-UT-04-05, ICL, Department of Computer Science, University of Tennessee, 2004.

[16] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:239–247, 1999.

[17] Xiao-Chuan Cai and Olof B. Widlund. Domain decomposition algorithms for indefinite elliptic problems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):243–258, 1992.

[18] S. Carney, M. Heroux, G. Li, and K. Wu. A revised proposal for a sparse blas toolkit, 1994.

[19] F. Cerioni, M. Colajanni, and S. Filippone ans S. Maiolatesi. PSBLAS user's guide. Technical Report RI.96.11, University of Rome Tor Vergata, 1996.

[20] Tony F. Chan and Tarek P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[21] J. Choi, J. Demmel, J. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers. *LAPACK Working Note*, 95, 1995.

[22] P. D'Ambra, D. di Serfaino, and S. Filippone. On the development of psblas-based parallel two-level schwarz preconditioners. Technical Report 1/2005, Dipartimento di Matematica. Seconda Universitá di Napoli, 2005.

[23] Tim A. Davis and Iain S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Software*, 25:1–19, 1999.

[24] J. Dongarra and R.C. Whaley. A user's guide to the BLACS v1.0. LAPACK Working Note #94 CS-95-281, University of Tennessee, June 1995. `http://www.netlib.org/lapack/lawns`.

[25] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *Int. J. High Perf. Comput. Appl.*, 17:125–131, 2003. also Lapack Working Note 157, ICL-UT-02-07.

[26] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[27] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of Basic Linear Algebra Subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, March 1988.

[28] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers.*

Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[29] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[30] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[31] E-Chow, R.D. Falgout, J.J. Hu, R.S. Tuminaro, and U. Meier Yang. A survey of parallelization techniques for multigrid solvers. Technical Report UCRL-BOOK-205864, August 2004.

[32] Evridiki Efstathiou and Martin J. Gander. Why restricted additive schwarz converges faster than additive schwarz. *BIT Numerical Mathematics*, 43(5):945–959, 2003.

[33] D. J. Evans. The use of pre-conditioning in iterative methods for solving linear equations with symmetric positive definite matrices. *J. Inst. Maths. Applics.*, 4:295–314, 1968.

[34] J.H. Ferziger and Milovan Peric. *Computational Methods for Fluid Dynamics*. Springer, 2002.

[35] S. Filippone, P. D'Ambra, and M. Colajanni. Using a parallel library of sparse linear algebra in a fluid dynamics applications code on linux clusters. In *Proceedings of ParCo*, 2001.

[36] S. Filippone, P. D'Ambra, and M. Colajanni. Using a parallel library of sparse linear algebra in a fluid dynamics applications code on linux clusters. *Parallel Computing - Advances & Current Issues*, pages 441–448, 2002.

[37] S. Filippone, M. Marrone, and G. Radicati di Brozolo. Parallel preconditioned conjugate-gradient type algorithms for general sparsity structures. *Intern. J. Computer Math.*, 40:159–167, 1992.

[38] S. Filippone, N. Rossi, G. Bella, and S. Ubertini. On the parallelization of the lattice-boltzmann method. In *Proceedings of PARA'04, to appear*, 2004.

[39] Salvatore Filippone and Michele Colajanni. PSBLAS: a library for parallel linear algebra computations on sparse matrices. *ACM Trans. on Math Software*, 26:527–550, 2000.

[40] Salvatore Filippone, Michele Colajanni, and Dario Pascucci. An object-oriented environment for sparse parallel computation on adaptive grids. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, page 365, Washington, DC, USA, 1999. IEEE Computer Society.

[41] O. Filippova and D. Hanel. Grid refinement for lattice-bgk models. *Journal of Computational Physics*, 147:219–228, November 1998.

[42] Andreas Frommer and Daniel B. Szyld. An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms. *SIAM Journal on Numerical Analysis*, 39:463–479, 2001.

[43] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[44] Anne Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[45] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2):452–469, 1995.

[46] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*.

[47] C. W. Hirt, A. A. Amsden, and J. L. Cook. An arbitrary lagrangian-eulerian computing method for all flow speeds. *J. Comput. Phys.*, 135(2):203–216, 1997.

[48] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[49] C.T. Kelley. *Iterative methods for Linear and Nonlinear Equations.* SIAM, 1995.

[50] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[51] Xiaoye S. Li. *Sparse Gaussian Eliminiation on High Performance Computers.* PhD thesis, University of California at Berkeley, 1996.

[52] L. Machiels and M. O. Deville. Fortran 90: an entry to object-oriented programming for the solution of partial differential equations. *ACM Trans. Math. Softw.*, 23(1):32–49, 1997.

[53] R. Mei, L. Luo, and W. Shyy. An accurate curved boundary treatment in the lattice boltzmann method. *Journal of Computational Physics*, 155, 1999.

[54] M. Metcalf and J. Reid. *Fortran 90 explained.* Oxford, 1990.

[55] M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. How fast are nonsymmetric matrix iterations. *SIAM J. Matrix Anal. Appl.*, 13(3):778–795, 1992.

[56] P.J. O'Rourke and A.A. Amsden. Implementation of a conjugate residual iteration in the kiva computer program. Technical Report LA-10849-MS, Los Alamos National Lab., 1986.

[57] P.J. O'Rourke, A.A. Amsden, and T.D. Butler. Kiva-ii: A computer program for chemically reactive flows with sprays. Technical Report LA-11560-MS, Los Alamos National Lab., 1989.

[58] Suhas V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Series in Computational Methods in Mechanics and Thermal Sciences. Taylor & Francis, first edition, 1980.

[59] G. Peng, H. Xi, G. Duncan, and S.H. Chou. Lattice boltzmann method on irregular meshes. *Physical Review*, 58(4):4124–4127, 1998.

[60] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of SuperComputing 99*, 1999.

[61] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, 1990.

[62] W.E. Pracht. Calculating three–dimensional fluid flows at all speeds with an eulerian–lagrangian computing mesh. *Journal of Computational Physics*, 17, 1975.

[63] J.W. Ruge and K. Stüben. Algebraic multigrid (amg), in multigrid methods, s.f. mccormick. *SIAM Frontiers on Applied Mathematics*, 3:73–130, 1987.

[64] Youseph Saad. *Iterative Methods for Sparse Linear Systems.* 2000.

[65] Barry F. Smith, Petter E. Bjørstad, and William D. Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, New York, NY, USA, 1996.

[66] D. Stevenson. A computational science manifesto, 1994.

[67] D. Stevenson. Software engineering frontiers in computational science and engineering, 1995.

[68] D. E. Stevenson. Science, computational science, and computer science: At a crossroads. In *ACM Conference on Computer Science*, pages 7–14, 1993.

[69] Klaus Stüben. A review of algebraic multigrid. Technical Report GMD-69, GMD-Forschungszentrum Informationstechnik GmbH, June 1999.

[70] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond.* Oxford University Press, 2001.

[71] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[72] L. N. Trefethen. Pseudospectra of linear operators. In Berlin Akademic-Verlag, editor, *Third International Congress on Industrial and Applied Mathematics*, 1995.

[73] Ray S. Tuminaro. Parallel smoothed aggregation multigrid: aggregation strategies on massively parallel machines. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*

(CDROM), page 5, Washington, DC, USA, 2000. IEEE Computer Society.

[74] A. M. Turing. Rounding-off errors in matrix processes. *Quart. J. of Mech. and Appl. Math.*, 1:287–308, 1948.

[75] S. Ubertini. Computational fluid dynamics through an unstructured lattice boltzmann scheme. In *Proceedings of IMECE 2003*, 2003. ASME paper no. IMECE2003-41194.

[76] S. Ubertini, G. Bella, and S. Succi. Lattice boltzmann method on unstructured grids: Further developments. *Physical Review*, 68, 2003.

[77] Petr Vanek, Jan Mandel, and Marian Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Technical Report UCD-CCM-036, 1, 1995.

[78] R. Vuduc, J. Demmel, and K. Yelikk. Oski: A library of automatically tuned sparse matrix kernels. In *(Proceedings of SciDAC 2005, Journal of Physics: Conference Series, to appear.*, 2005.

[79] Richard W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California Berkeley, 2003.

[80] P. Wesseling. *An Introduction to Multigrid Methods*. John Wiley & Sons, Chichester, 1992.

[81] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[82] SIAM working groups on CSE education. Graduate education in computational science and engineering. In *SIAM Review*, pages 163–177, 2001.

[83] http://math.nist.gov/MatrixMarket/.

[84] http://www.enseeiht.fr/lima/apo/MUMPS/.

[85] http://www.cise.ufl.edu/research/sparse/matrices/.

[86] http://www.cise.ufl.edu/research/sparse/umfpack/.

Appendix A

Experimental Setup

Table A.2 contains the set of matrices that have been used to test and tune the automatic selection procedure of the BCSR storage format block size presented in chapter 3. Matrices are reported along with their size, number of nonzero elements and nonzero elements per row which has, as described in section 3.3.2, a deep impact on the flop rate of the sparse matrix-vector product operation. All of the matrices have been downloaded from "University of Florida Sparse Matrix Collection" [85] and the "Matrix Market" [83].

Tables A.3 and A.4 contain the architecture that have been used to test and tune the automatic selection procedure of the BCSR storage format block size presented in chapter 3. Details are given about the kind of processor and its clock frequency, the cache size, the memory size, the Operating System and the compilers used. The AMD Athlon 1200, the Itanium2, the MIPS and Power3 architectures are installed at the Innovative Computing Laboratory of the University of Tennesse Knoxville (UTK). The AMD Athlon 1800, the PentiumIII and the Xeon machines are installed at the CE Laboratory of the "Tor Vergata University of Rome", Computer Science Engineering department. The AMD Athlon 64-bit 3500+ is installed at the Mechanics Engineering department of the "Tor Vergata" University of Rome.

Matrix	Dimension	Nonzero	Nonzeroes per row
west2021.mtx	2021 X 2021	7353	3.64
sherman3.mtx	5005 X 5005	20033	4.00
shyy161.mtx	76480 X 76480	329762	4.31
finan512.mtx	74752 X 74752	335872	4.49
bayer02.mtx	13935 X 13935	63679	4.57
pwt.mtx	36519 X 36519	181313	4.96
zenios.mtx	2873 X 2873	15032	5.23
jpwh_991.mtx	991 X 991	6027	6.08
saylr4.mtx	3564 X 3564	22316	6.26
onetone2.mtx	36057 X 36057	227628	6.31
orsreg_1.mtx	2205 X 2205	14133	6.41
lns_3937.mtx	3937 X 3937	25407	6.45
lnsp3937.mtx	3937 X 3937	25407	6.45
gemat11.mtx	4929 X 4929	33185	6.73
wang3.mtx	26064 X 26064	177168	6.80
wang4.mtx	26068 X 26068	177196	6.80
bayer10.mtx	13436 X 13436	94926	7.07
memplus.mtx	17758 X 17758	126150	7.10
vibrobox.mtx	12328 X 12328	177578	14.40
s3rmt3m1.mtx	5489 X 5489	112505	20.50
bai.mtx	23560 X 23560	484256	20.55

Table A.1: Details about the matrices used to tune and test te performance model presented in chapter 3

Matrix	Dimension	Nonzero	Nonzeroes per row
coater2.mtx	9540 X 9540	207308	21.73
lhr10.mtx	10672 X 10672	232633	21.80
rdist1.mtx	4134 X 4134	94408	22.84
bcsstm27.mtx	1224 X 1224	28675	23.43
bcsstk35.mtx	30237 X 30237	740200	24.48
nasasrb.mtx	54870 X 54870	1366097	24.90
ct20stif.mtx	52329 X 52329	1375396	26.28
venkat01.mtx	62424 X 62424	1717792	27.52
mcfe.mtx	765 X 765	24382	31.87
gupta1.mtx	31802 X 31802	1098006	34.53
crystk02.mtx	13965 X 13965	491274	35.18
orani678.mtx	2529 X 2529	90158	35.65
3dtube.mtx	45330 X 45330	1629474	35.95
crystk03.mtx	24696 X 24696	887937	35.95
vavasis3.mtx	41092 X 41092	1683902	40.98
goodwin.mtx	7320 X 7320	324784	44.37
rim.mtx	22560 X 22560	1014951	44.99
olafu.mtx	16146 X 16146	1015156	62.87
ex11.mtx	16614 X 16614	1096948	66.03
raefsky4.mtx	19779 X 19779	1328611	67.17
raefsky3.mtx	21200 X 21200	1488768	70.22
mbeaflw.mtx	496 X 496	49920	100.64
dense.mtx	1500 X 1500	2250000	1500.00

Table A.2: Details about the matrices used to tune and test te performance model presented in chapter 3

	AMD Athlon 1200	AMD Athlon 1800	AMD Athlon 64-bit 3500+	Itanium2
Proc. type	AMD Athlon k6	AMD Athlon MP 2200+	AMD Athlon 64 Processor 3500+	Genuine Intel IA-64 Itanium2
Proc. freq.	1200 MHz	1800 MHz	2211 MHz	900 MHz
Cache size	64 KB L1 256 KB L2	64 KB L1 256 KB L2	64 KB L1 512 KB L2	32 KB L1 256 KB L2 1.5 MB L3
Memory size	256 MB	2 GB	2 GB	8 GB
OS	GNU-Linux	Fedora Core 3	Fedora Core 3	Red Hat Linux 3.2.3
Compilers	Intel compilers v9.0	Intel compilers v9.0	Intel Compilers v9.0	Intel Compilers v9.0

Table A.3: Details of the architectures used to test and tune the performance model presented in chapter 3

	MIPS	PentiumIII	Power3	Xeon
Proc. type	MIPS R12000	PentiumIII (Coppermine)	IBM Power3	Intel Xeon
Proc. freq.	270 MHz	930 MHz	375 MHz	3057 MHz
Cache size	32 KB L1 2 MB L2	16 KB L1 256 KB L2	64 KB L1 8 MB L2	32 KB L1 512 KB L2
Memory size	256 MB	512 MB	1 GB	1 GB
OS	IRIX64 6.5	Fedora Core 3	AIX 5.1	Fedora Core 3
Compilers	MIPSpro Compilers v7.41	Intel Compilers v9.0	IBM xlc and IBM xlf v6.0	Intel Compilers v9.0

Table A.4: Details of the architectures used test and tune the performance model presented in chapter 3