Chapter 4

Synchronous Iterations

Introduction

In this chapter, we are interested in parallel synchronous iterative algorithms for linear and nonlinear systems. Convergence results of the synchronous versions and their implementations are detailed.

We will concentrate on so-called *multisplitting algorithms* and their coupling with the Newton method. Multisplitting algorithms include the discrete analogues of Schwarz multi-subdomain methods and hence are very suitable for distributed computing on distant heterogeneous clusters. They are particularly well suited for physical and natural problems modeled by elliptic systems and discretized by finite difference methods with natural ordering.

The parallel versions of minimization like the methods exposed in Chapter 2 are not detailed in this chapter but it should be mentioned that, thanks to the multisplitting approach and under suitable assumptions on the splittings, these methods can be used as *inner iterations* of *two-stage* multisplitting algorithms.

4.1 Parallel linear iterative algorithms for linear systems

4.1.1 Block Jacobi and O'Leary and White multisplitting algorithms

Suppose that we have L processors $P_1, ..., P_L$ and that an unknown vector of dimension n is partitioned into L subvectors of dimensions n_i ($i \in \{1, ..., L\}$) so that $n = \sum_{i=1}^{L} n_i$, $\mathbb{R}^n = \prod_{i=1}^{L} \mathbb{R}^{n_i}$.

Consider the n-dimensional linear system

$$Ax = b, \; x \in \mathbb{R}^n, \tag{4.1}$$

and suppose that (4.1) has a unique solution x^*.

As seen in Section 2.1.6 of Chapter 2, block iterative algorithms can be deduced from by-point iterative algorithms by splitting the matrix A into $M - N$

where M and N are block matrices. The parallel block Jacobi algorithm consists in taking M as a block nonsingular diagonal matrix

$$M = D = \begin{pmatrix} A_{11} & 0 & \cdots & \cdots & 0 \\ 0 & A_{22} & \ddots & & \vdots \\ \vdots & \ddots & A_{33} & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & A_{nn} \end{pmatrix}, \quad N = \begin{pmatrix} 0 & -A_{12} & -A_{13} & \dots & -A_{1n} \\ -A_{21} & 0 & -A_{23} & \dots & -A_{2n} \\ -A_{31} & -A_{32} & 0 & \dots & -A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -A_{n1} & -A_{n2} & -A_{n3} & \dots & 0 \end{pmatrix} \tag{4.2}$$

where A_{ii} are matrices of dimension $n_i \times n_i$, so that at each iteration k, each processor P_i solves for $X_i^{(k+1)}$ the linear system

$$A_{ii}X_i^{(k+1)} = -\sum_{j\neq i} A_{ij}X_j^{(k)} + B_i,$$

where X_i and B_i are the i^{th} block components of x and b of dimensions $n_i \times 1$ so that we have

$$x = (X_1, ..., X_L)^T \text{ and } b = (B_1, ..., B_L)^T.$$

The convergence of the parallel block Jacobi algorithm is deduced from Theorem 2.1 of Section 2.1 of Chapter 2. Indeed, it is sufficient to consider that the new fixed point mapping T is the one corresponding to the block Jacobi matrix $J = M^{-1}N$ where M and N are defined above in (4.2).

In this section we introduce O'Leary and White algorithms ([92], [118]) which generalize the parallel block Jacobi algorithms. Let us first recall some definitions and results which will be helpful in the comparison of the speed of convergence of the different forthcoming parallel algorithms.

DEFINITION 4.1 *We say that a vector x is nonnegative (positive), denoted $x \geq 0$ $(x > 0)$, if all its entries are nonnegative (positive). A matrix B is said to be nonnegative, denoted $B \geq 0$, if all its entries are nonnegative. We compare two matrices $A \geq B$, when $A - B \geq 0$, and two vectors $x \geq y$ $(x > y)$ when $x - y \geq 0$ $(x - y > 0)$.*

DEFINITION 4.2 *Let A be a $n \times n$ real matrix. The decomposition $A = M - N$ is called a splitting if M is nonsingular. It is called a convergent splitting if $\rho(M^{-1}N) < 1$. A splitting $A = M - N$ is called:*
(a) regular if $M^{-1} \geq 0$ and $N \geq 0$
(b) weak regular if $M^{-1} \geq 0$ and $M^{-1}N \geq 0$.

THEOREM 4.1
Let $A = M - N$, where A and M are nonsingular square matrices. Let $T = M^{-1}N$ and suppose that T is a nonnegative matrix, then

$$\rho(T) < 1 \Leftrightarrow A^{-1}N \geq 0.$$

Moreover

$$\rho(T) = \frac{\rho(A^{-1}N)}{1 + \rho(A^{-1}N)}.$$

PROOF Suppose that $\rho(T) < 1$. Then

$$\begin{aligned} A^{-1}N &= \left[M(I - M^{-1}N)\right]^{-1} N \\ &= (I - T)^{-1}T \\ &= \textstyle\sum_{p=1}^{\infty} T^p. \end{aligned}$$

As T is nonnegative, we deduce that $A^{-1}N$ is nonnegative.

Suppose now that $A^{-1}N \geq 0$. Then the Perron-Frobenius theorem implies that there exists a positive vector such that

$$Tx = \rho(T)x.$$

So

$$\begin{aligned} A^{-1}Nx &= (I - T)^{-1}Tx \\ &= \frac{\rho(T)}{1 - \rho(T)}x. \end{aligned}$$

As $A^{-1}N$ and x are nonnegative, the last equality implies that $\rho(T) < 1$. Now, the equation just quoted above implies that

$$\frac{\rho(T)}{1 - \rho(T)} \leq \rho(A^{-1}N),$$

hence

$$\rho(T) \leq \frac{\rho(A^{-1}N)}{1 + \rho(A^{-1}N)}.$$

On the other hand, as $A^{-1}N \geq 0$, we have by the Perron-Frobenius theorem

$$\begin{aligned} Ty &= (I + A^{-1}N)^{-1}A^{-1}Ny \\ &= \frac{\rho(A^{-1}N)}{1 + \rho(A^{-1}N)}y, \end{aligned}$$

for some positive vector y, thus

$$\rho(T) \geq \frac{\rho(A^{-1}N)}{1 + \rho(A^{-1}N)}.$$

$\square$

THEOREM 4.2
Let $A = M - N$ be a weak regular splitting of A. Then the following assertions are equivalent:

1. $A^{-1} \geq 0$.
2. $A^{-1}N \geq 0$.
3. $\rho(T) < 1$.

PROOF 1) implies 2) since $A^{-1}N = (I-T)^{-1}T = \sum_{p=1}^{\infty} T^p$ (Neumann Lemma, see the Appendix).
2) $\Leftrightarrow$ 3) by Theorem 4.1.
3) $\Rightarrow$1) : since $A^{-1} = (I-T)^{-1}M^{-1} = \sum_{p=1}^{\infty} T^p M^{-1}$. ∎

PROPOSITION 4.1
Consider a square $n \times n$ matrix A such that $A^{-1} \geq 0$. Let

$$A = M_1 - N_1 = M_2 - N_2$$

be two regular splittings of A. Denote by $T_1 = M_1^{-1}N_1$ and by $T_2 = M_2^{-1}N_2$, then

$$N_2 \leq N_1 \Rightarrow \rho(T_2) \leq \rho(T_1),$$

so

$$R_\infty(T_1) \leq R_\infty(T_2).$$

PROOF This is a consequence of Theorem 4.2 and the fact that the function $f(x) = x/(1+x)$ is monotone increasing. ∎

The above results allow us to compare two block Jacobi like algorithms. Indeed, the decompositions $A = M_1 - N_1$ and $A = M_2 - N_2$ give rise to the block Jacobi algorithms whose iteration matrices are, respectively, $M_1^{-1}N_1$ and $M_2^{-1}N_2$. Indeed, the behaviors of synchronous iterations, generated by these block Jacobi algorithms to solve the linear system (4.1), are, respectively, described by the successive approximations associated to the fixed point mapping

$$\begin{aligned} T^{(1)} : \mathbb{R}^n &\to \mathbb{R}^n \\ x &\mapsto y = M_1^{-1}N_1x + M_1^{-1}b \end{aligned}$$

and

$$\begin{aligned} T^{(2)} : \mathbb{R}^n &\to \mathbb{R}^n \\ x &\mapsto y = M_2^{-1}N_2x + M_2^{-1}b. \end{aligned}$$

Then, Theorems 4.1 and 4.2 give sufficient conditions to ensure the convergence of block Jacobi algorithms. Theorem 4.1 allows us to compare the speed of convergence of two given block Jacobi algorithms.

The multisplitting approach consists in partitioning the matrix A into horizontal band matrices. Then each processor, or group of processors, is responsible for the management of a band matrix and the associated unknown subvector of x. Multisplitting methods were first introduced by O'Leary and White in [92], [118], [116]; they define a multisplitting of A as a collection of L triplets (B_l, C_l, D_l) such that

1. $A = B_l - C_l$, for $l = 1, \dots, L$ where B_l is nonsingular.
2. $\sum_l D_l = I$ where D_l $(l = 1, ..., L)$ are diagonal nonnegative matrices and I is the identity matrix.
   Then the multisplitting algorithm is defined as follows ($x^{(0)}$ given):

Algorithm 4.1 Multisplitting scheme

```
for i=0,1,..., until convergence do
  for l=1,...,$L$ do
    $y_l \leftarrow B_l^{-1} C_l x^{(i)} + B_l^{-1} b$
  end for
  $x^{(i+1)} \leftarrow \sum_l D_l y_l$
end for
```

O'Leary and White [92] established the following result:

THEOREM 4.3
If for $l = 1, ..., L$, (B_l, C_l) are weak regular splittings of A satisfying $A^{-1} \geq 0$, then Algorithm 4.1 is convergent.

The convergence of O'Leary and White multisplitting algorithms given in Theorem 4.3 is based on Theorem 4.1. It can be seen that block Jacobi algorithms correspond to the particular case of the O'Leary and White multisplitting method where the matrix is partitioned into non-overlapping blocks and where the entries of the weighted diagonal matrices are null when they are not associated with the computation of the vector associated with the block diagonal matrix.

Since the work of O'Leary and White, several authors have studied multisplitting algorithms for linear and nonlinear systems; we refer to [59], [61], [60], [75], [27], [62], [5] and the references therein.

In the next section, we give a general formulation of multisplitting algorithms due to Bahi et al. [27]. This formulation allows us to put in the same theoretical framework, parallel block Jacobi algorithms, O'Leary and White multisplitting algorithms and the discrete analogues of parallel Schwarz algorithms.

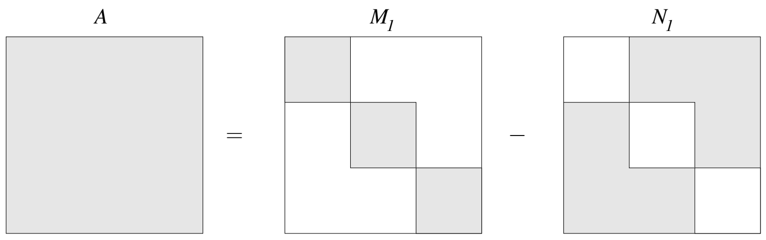

FIGURE 4.1: A splitting of matrix A.

4.1.2 General multisplitting algorithms

In this section we follow the general formulation of multisplitting algorithms given in [27]. These algorithms are described by the iterations generated by the successive approximations associated to an *extended fixed point mapping* defined from $(\mathbb{R}^n)^L$ into itself, where n is the dimension of the problem and L is the number of processors. This fixed point mapping is defined as follows:

$$\left\{\begin{array}{rcl} \mathcal{T} : (\mathbb{R}^n)^L & \longrightarrow & (\mathbb{R}^n)^L \\ X = (x^1, ..., x^L) & \longmapsto & Y = (y^1, ..., y^L), \end{array}\right. \tag{4.3}$$

such that for $l \in \{1, ..., L\}$

$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum\limits_{k=1}^{L} E_{lk} x^k, \end{cases} \tag{4.4}$$

where E_{lk} are weighting matrices satisfying

$$\begin{cases} E_{lk} \text{ are diagonal matrices} \\ E_{lk} \geq 0 \\ \sum\limits_{k=1}^{L} E_{lk} = I_n \quad (\text{identity matrix}), \quad \forall l \in \{1, ..., L\}. \end{cases} \tag{4.5}$$

In (4.4),

$$T^{(l)}(z^l) = M_l^{-1} N_l z^l + M_l^{-1} b \tag{4.6}$$

where

$$A = M_l - N_l, \quad l = 1, ..., L \tag{4.7}$$

is a splitting of A and M_l is, e.g., the block diagonal matrix defined in Figure 4.1.

Then it can be shown that if each splitting is convergent, i.e., if $\rho(M_l^{-1} N_l) < 1$, then the extended fixed point mapping is also convergent to the extended solution of (4.1), say $(x^*, ..., x^*)$, and then the synchronous algorithm converges. The convergence study will be detailed in the

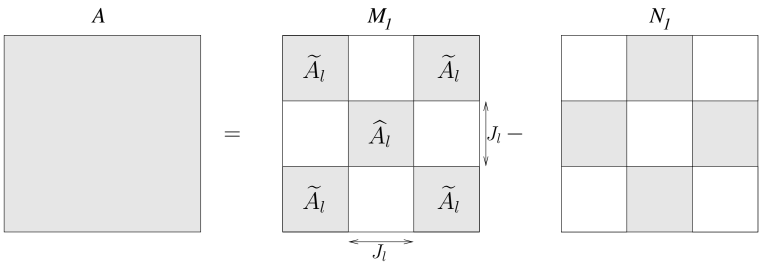

FIGURE 4.2: A splitting of matrix A using subset J_l of $l \in \{1, ..., L\}$.

next chapter in the more general context which includes the study of both synchronous and asynchronous algorithms.

In the following, a matrix A is partitioned as follows:

$$\begin{aligned} \left(\widehat{A}_l\right)_{i,j} &= a_{i,j}, \ for \ i, \ j \in J_l, \\ \left(\widetilde{A}_l\right)_{i,j} &= a_{i,j} \ for \ i, \ j \in J_l^C, \\ M_l &= diag\left(\widehat{A}_l, \widetilde{A}_l\right), \end{aligned}$$

where L denotes the number of processors and J_l are subsets of $\{1, ..., n\}$. The elements of J_l are indices of sub-components of a vector x of $\mathbb{R}^n$. To each $l \in \{1, ..., L\}$ we associate a splitting, so we obtain L splittings of A described in Figure 4.2.

We will now show how the extended fixed point defined above and the dependence of the weighting matrices on both l and k allow us to obtain particular standard algorithms such as O'Leary and White multisplitting algorithms.

The practical considerations on how to implement such algorithms are discussed in Section 4.4.5.

4.1.2.1 Obtaining O'Leary and White multisplitting

If the diagonal positive matrices E_{lk} depend only on k

$$E_{lk} = E_k$$

and satisfy

$$\begin{cases} \sum_{k=1}^{L} E_k = I_n \\ (E_k)_{i,i} = 0, \ \forall i \notin J_k \end{cases} \tag{4.8}$$

Then the synchronous iterations corresponding to O'Leary and White multisplitting are defined by the fixed point mapping (here $L = n$, $B_l = \mathbb{R}^n$),

$$\mathcal{T}^{OW}(x^1, ..., x^L) = (y^1, ..., y^L) \ such \ that$$

$$\begin{cases} y^l = T^{(l)}(z) \\ z = \sum\limits_{k=1}^{L} E_k x^k \end{cases}$$

where for $l \in \{1, ..., L\}$, $T^{(l)}$ is defined by (4.6).

4.1.2.2 Obtaining discrete analogues of Schwarz alternating algorithms

Suppose that we have only two subsets J_1 and J_2 and that $J_1 \bigcap J_2 \neq \emptyset$, so we have an overlapping between the 1st and the 2nd subdomains and

$$A = M_1 - N_1 = M_2 - N_2$$

Consider the matrices E_{lk} such that

$$(E_{11})_{i,i} = \begin{cases} 1 \ \forall i \in J_1 \\ 0 \ \forall i \notin J_1 \end{cases}, \quad (E_{12})_{i,i} = \begin{cases} 0 \ \forall i \in J_1 \\ 1 \ \forall i \notin J_1 \end{cases} \tag{4.9}$$

$$(E_{21})_{i,i} = \begin{cases} 1 \ \forall i \notin J_2 \\ 0 \ \forall i \in J_2 \end{cases}, \quad (E_{22})_{i,i} = \begin{cases} 0 \ \forall i \notin J_2 \\ 1 \ \forall i \in J_2 \end{cases}$$

Define the fixed point mapping

$$\mathcal{T}^S(x^1, x^2) = (y^1, y^2) \ such \ that \ for \ l = 1, 2$$

$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum\limits_{k=1}^{2} E_{lk} x^k \end{cases} \tag{4.10}$$

where for $l \in \{1, 2\}$, $T^{(l)}$ is defined by (4.6). Then the additive discrete analogue of the Schwarz alternating method corresponds to the successive approximation method applied to $\mathcal{T}^S$, and the multiplicative discrete analogue of the Schwarz alternating method corresponds to the block nonlinear Gauss-Seidel method applied to $\mathcal{T}^S$.

4.1.2.3 Obtaining discrete analogues of multisubdomain Schwarz algorithms

We introduce the weighting matrices E_k satisfying (4.8) and the matrices E_{lk} such that for $l \in \{1, ..., L\}$

$$\begin{aligned} (E_{ll})_{i,i} &= \begin{cases} 1 \ if \ i \in J_l \\ 0 \ if \ i \notin J_l \end{cases} \\ (E_{lk})_{i,i} &= \begin{cases} 0 \ if \ i \in J_l \\ (E_k)_{i,i} \ \ if \ i \notin J_l \end{cases} \end{aligned} \tag{4.11}$$

the synchronous iterations corresponding to the discrete analogue of the multisubdomain Schwarz method are defined by the fixed point mapping $\mathcal{T}^{MS}$

$$\mathcal{T}^{MS}(x^1, ..., x^L) = (y^1, ..., y^L) \ such \ that$$

$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum_{k=1}^{L} E_{lk} x^k \end{cases} \tag{4.12}$$

where E_{lk} are defined by (4.11) and $T^{(l)}$ are defined by (4.6).

4.1.2.4 Convergence of multisplitting and *two-stage* multisplitting algorithms

As mentioned above, the convergence of synchronous multisplitting algorithms was established by O'Leary and White in [92] when $A^{-1} \geq 0$, where A is the matrix of the linear system, and the splittings are weak regular.

When the linear systems, arising from the multisplitting algorithm, are not solved exactly but are instead approximated by iterative methods, we are confronted with *two-stage multisplitting algorithms* or *inner-outer iterations.* The convergence of two-stage multisplitting algorithms when the number of inner iterations is fixed was established by Szyld and Jones in [109] (see also [75]) for $A^{-1} \geq 0$ when the outer splittings are regular and the inner splittings are weak regular. The convergence of two-stage multisplitting algorithms was also studied by Bahi et al. [27] in a more general context including linear and nonlinear systems of equations.

4.2 Nonlinear systems: parallel synchronous Newton-multisplitting algorithms

Now we are interested in the development of parallel algorithms for nonlinear problems. We concentrate on the Newton method, since it is the most commonly used method to solve nonlinear systems.

4.2.1 Newton-Jacobi algorithms

Consider the nonlinear problem

$$F(x) = 0 \tag{4.13}$$

and the Newton method defined in Chapter 2 by the iterations

$$x^{(k+1)} = x^{(k)} - F'(x^{(k)})^{-1} F(x^{(k)}).$$

The solution of the system

$$F'(x^{(k)})x^{(k+1)} = F'(x^{(k)})x^{(k)} - F(x^{(k)}) \tag{4.14}$$

may be particularly prohibitive when the dimension of the problem is large. In this case, we can use an iterative method instead of direct ones in order to obtain an approximate solution of the system (4.14).

Let $D^{(k)}$ be a block diagonal matrix of $F'(x^{(k)})$, then

$$F'(x^{(k)}) = D^{(k)} - D^{(k)} + F'(x^{(k)}). \tag{4.15}$$

The linear system (4.14) can be solved by the block Jacobi algorithm associated with the splitting (4.15). The obtained scheme consists in computing the iteration vectors by the following *two-stage* algorithm.

Algorithm 4.2 Newton-Jacobi scheme

```
Choose any arbitrary initial vector $(x^{(0)})^{(0)} = 0$
for k = 1,2,... do
  for l = 1,2,... do
    $D^{(k)}(x^{(k+1)})^{(l+1)} \leftarrow (D^{(k)} - F'(x^{(k)}))(x^{(k)})^{(l)} - F(x^{(k)})$
  end for
end for
```

It should be noticed that in practice only a fixed number of inner iterations is performed and that the number of inner iterations may vary in function of the Newton outer iterations. We are then in the presence of nonstationary iterative methods. The next section introduces Newton-multisplitting algorithms which are a generalization of Newton-Jacobi algorithms.

4.2.2 Newton-multisplitting algorithms

We suppose that (4.13) has a solution x^* and that F is Fréchet differentiable on a neighborhood of x^*. We also suppose that F' is nonsingular and Lipschitz continuous on a neighborhood of x^*. Newton iterations can be rewritten in the form

$$x^{(k+1)} = x^{(k)} - y^{(k)},\ k = 0, 1, 2, ...$$

where $y^{(k)}$ is the solution of the linear system

$$F'(x^{(k)})y = F(x^{(k)}) \tag{4.16}$$

Using an iterative method to solve (4.16) gives rise to the so-called *Newton iterative methods* [5], [6]. In [117], White proposes the parallel Newton-SOR method in order to solve nonlinear systems on parallel computers. In [5] and [6], the authors propose nonstationary multisplitting methods to solve (4.16), i.e., they consider for each k, a collection of L splittings of $F'(x^{(k)})$,

$$F'(x^{(k)}) = M_l(x^{(k)}) - N_l(x^{(k)}),\ l = 1, ..., L, \tag{4.17}$$

Suppose that the weighting matrices (4.5) only depend on one index and that the solution of system (4.16) is approximated by performing q iterations of the multisplitting method and that $y^{(0)} = 0$.

The parallel Newton-multisplitting method can be written as follows:

$$x^{(k+1)} = G(x^{(k)}), \tag{4.18}$$

where

$$G(x) = x - A(x)F(x),$$

and

$$A(x) = \sum_{l=1}^{L} E_l(x)(I - (M_l(x)^{-1}N_l(x))^q(F'(x))^{-1}. \tag{4.19}$$

The following result gives the convergence condition of synchronous Newton-multisplitting algorithms.

THEOREM 4.4
If the splittings (4.17) are convergent, then there exists a neighborhood V_{x^*} of the solution x^*, such that any synchronous Newton-multisplitting algorithm associated with (4.18) and (4.19) starting from $x^{(0)} \in V_{x^*}$ converges to x^*.

PROOF We have

$$G'(x^*) = I - A(x^*)F'(x^*). \tag{4.20}$$

From (4.19) we have

$$G'(x^*) = I - \sum_{l=1}^{L} E_l(x^*)(I - (M_l(x^*)^{-1}N_l(x^*))^q. \tag{4.21}$$

The properties of the weighting matrices imply that

$$G'(x^*) = \sum_{l=1}^{L} E_l(x^*)(M_l(x^*)^{-1}N_l(x^*))^q. \tag{4.22}$$

As the splittings (4.17) are convergent, we deduce by the application of proposition 3.2 of [27] that

$$\rho(G'(x^*)) \leq \max_{1 \leq l \leq L} \rho((M_l(x^*)^{-1}N_l(x^*))^q) < 1.$$

The result follows from Ostrowski theorem [95] (see the Appendix). □

One can also suppose that the approximate solution of (4.16) is done by performing different $q_{k,l}$ inner linear iterations based on the linear splittings as explained in [5]. The obtained two-stage algorithm is called a *nonstationary Newton iterative algorithm.* The following convergence result is proved in [6].

THEOREM 4.5

If any of the following two conditions is satisfied, then there exists a neighborhood V_{x^*} such that the Newton-multisplitting started with $x^{(0)} \in V_{x^*}$ converges to x^*,

1. *$F'(x^*)$ is monotone and the splittings (4.17) are weak regular.*
2. *$F'(x^*)$ is an H-matrix and the splittings (4.17) are H-compatible (see the Appendix).*

4.3 Preconditioning

Some preconditioning algorithms have been adapted to parallel synchronous algorithms. In this case, depending on the amount of communications and of synchronizations, on the granularity of the method, on the degree of parallelism and especially on the network efficiency, the performances of those algorithms are relatively limited with a large number of processors. Nevertheless, some preconditioners have been designed for parallel architectures. For example, parallel preconditioners, based on ILU, are very sensitive to the ordering of the unknowns. The more independent the unknowns are, the more efficient the parallel preconditioner ILU is. For more explanations on parallel preconditioners, interested readers are invited to read [29, 102, 121, 41] and the references therein.

4.4 Implementation

Implementing a synchronous parallel algorithm depends on the platform used to execute it. In fact, it is possible to distinguish at least two different paradigms from the programming point of view. The first one is only dedicated to shared memory architectures. The second one is commonly called *message passing* and is mainly used in distributed architectures.

On shared memory architectures, at least two different kinds of programming exist. The first class aims at parallelizing most consuming loops. Consequently we obtain what is usually called a *data parallel* code which fits the class of fine grained parallelism. In this model, the same set of instructions runs simultaneously on different pieces of data. More precisely, each processor executes only a part of the loop. The other parts are achieved by other processors. In this context, the best-known programming model is certainly OpenMP [36] for which a programmer only needs to add compiler directives in its code. In some particular cases the programmer does not need to modify its code. Compiler directives are interpreted in order to split the initial work of loops into smaller parts that are executed by the available processors.

The second class aims at decomposing the work into large parts with smaller parts in which processors exchange some data. Usually this method is efficient if the parts are relatively independent. This model is frequently called *coarse grained* parallelism.

In both these models, processors can access to the whole memory for reading and writing. If two processors access the same data in writing, the behavior is often nondeterministic. So the programmer must carefully check that only one processor writes into a part of data at each instant. However, the big advantage of this model is its programming simplicity since a processor can directly read any part of the memory without asking it to any processor. Of course according to the architecture, the time to read data is not always constant and this often leads to bottlenecks.

Concerning the programming of synchronous iterative algorithms both models are interesting but do not provide the same programming work. Using fine grained parallelism with loops splitting is quite easy. Nevertheless such codes are not as scalable as coarse grained parallelism codes which require a longer programming endeavor. Now, most programmers do agree with the fact that the transformation of a sequential program into a parallel one using shared memory mechanism requires less work than using other parallelization paradigms. Moreover, as few architectures provide a shared memory mechanism, an application parallelized using that paradigm would not be as portable as if it were parallelized using another programming model.

An interesting alternative, if we are interested in code reuse, lies in designing an application with the message passing paradigm. This is the classical model used for distributed architectures in which processors communicate by sending/receiving messages to/from each other. Using a message passing paradigm often requires rather a lot more work and time to design a parallel application compared to using a shared memory paradigm. Nonetheless, such a program is more portable since it can be executed on many architectures: either distributed ones or shared memory ones. Even if it is not as efficient on shared memory architecture as using a shared memory paradigm, it is possible to run a program designed with a message passing paradigm on such an architecture. Message passing programs generally require the use of buffers in order to send or receive messages; that is why on shared memory architectures, they could be less efficient.

With the development of multi-core processors, scientists have access to clusters in which both paradigms can be used in order to benefit from the best performance. On the one hand, communications between processors linked by a network should be achieved using a message passing interface. On the other hand, inside a multiprocessor or a multi-core machine, a shared memory paradigm is preferable to obtain efficient codes. So, in this kind of architecture, which will probably be used more and more in the next years, the mixing of shared memory and message passing paradigms will probably be the best solution in order to obtain efficient codes.

Designing a parallel algorithm to solve a linear system or a nonlinear one requires approximately the same notions from the programming point of view. First it is extremely important to be rigorous. Of course this is important for implementing any sequential algorithm but it is more important as far as parallel application is concerned, the least error being indeed extremely difficult to detect. Implementing a parallel algorithm depends on the parallel language chosen. Nevertheless, with some experience, it is relatively easy to disregard it. This is quite similar to implementing a sequential algorithm with one language or with another one which only differs by the syntax. That is why in the following we will only focus on a general implementation that will slightly differ from using MPI with the language C or OpenMP with C++ and a coarse-grained paradigm.

4.4.1 Survey of synchronous algorithms with shared memory architecture

As shared memory architectures are not as scalable as distributed ones, in this book, we only describe the principles for designing parallel synchronous iterative algorithms using this kind of architecture. Moreover, parallelizing an application using a shared memory architecture is often easier than using a distributed one.

Since the parallelization of an iterative algorithm using the coarse grained paradigm with a shared memory architecture is quite similar to the parallelization of the same algorithm with a distributed system, we only focus on the fine grained paradigm. In addition, this paradigm is easier to implement and is probably the most used model with a limited number of processors, as it is the case with cheaper architectures (small multi-processor or multi-core systems).

Since OpenMP is the most used tool to build fine grained parallel algorithms, we limit our explanation to it. Roughly speaking, iterative algorithms are all composed of a loop which represents an iteration. As an iteration uses the results computed in previous iterations it is not possible to execute different iterations concurrently on different processors. Therefore, it is necessary to parallelize the computation inside an iteration. Consequently, the number of iterations with such a parallel program will be exactly the same as the sequential code used for the parallelization. The fact of using the fine grained paradigm with OpenMP consists in parallelizing all the loops inside an iteration as soon as the work inside a loop can be done concurrently. It should be noticed that several loops can be parallelized inside an iterative algorithm. For example, scalar products, matrix-vector products, or vector additions can be executed using parallelization at the loop level. Taking the Jacobi algorithm (Algorithm 2.1), it is possible to parallelize the first loop indexed by i. Concerning the Gauss-Seidel algorithm (Algorithm 2.2), it is well-known that this algorithm is less parallelizable. It is not possible to parallelize the same loop. Nevertheless it is possible to parallelize the inner loops, indexed by j in that algorithm. Of course, this parallel scheme is less efficient since the parallelization of small loops only provides a small performance gain. The parallelization of nonlinear methods is sensibly similar. Only some loops inside an iteration are parallelizable. Practically speaking, this kind of parallelization provides good speed-ups with few processors, but unfortunately, bad speed-ups as soon as the number of processors increases.

Coarse grained parallelism implementation is completely different and uses approximately the same scheme either with a shared memory architecture or with a distributed one (except for the communication handling). In the following sections we distinguish the case of sequential algorithms that can be parallelized using traditional message passing schemes from the Jacobi algorithm and multisplitting ones for which an appropriate implementation will enable them to be executed in an asynchronous mode as we will see in Chapter 5.

4.4.2 Synchronous Jacobi algorithm

It is well known that it is generally difficult to implement an efficient and general code. According to the structure of the studied matrix, some optimizations can produce very efficient codes. In the following we give an implementation of the Jacobi parallel code. In order to keep this algorithm simple, we consider that a processor needs to send its results to all other processors. In many cases, it is easy to compute the list of neighbors for each processor and, consequently, only send the results to processors that require it. So, we consider that the matrix A is split into rectangular parts as in Figure 4.3. The vector B is split as the vector X.

Each processor only owns a part of the vector X but it requires a larger part of the vector $XOld$. According to the structure of the matrix A, this part may vary. If A is an almost dense matrix, then each processor needs the totality of the vector $XOld$. Algorithm 4.3 illustrates the synchronous Jacobi algorithm. In this algorithm *Size* and *SizeGlo* represent, respectively, the local size and the global size of the matrix A, *Offset* represents the offset of the global index for the computation, i.e., the sum of the local size of all parts of matrices of processors having a lower rank. The variable $MyRank$ represents the rank of the processor, that is to say the number of the current processor in the computation. In order to detect the convergence, we first compute at each iteration the local convergence and put the result in the variable $Error$. Then we use a reduce operation that computes the maximum of all the local errors. With MPI, such an operation is directly implemented in the API.

Without considering the communications, this parallel version of the Jacobi method is completely similar to the sequential one. In the algorithm we are using a high level communication procedure called $AllReduce$. The goal of this procedure consists in applying a reduction operation on the variable $Error$

Algorithm 4.3 Synchronous Jacobi algorithm

```
NbProcs: number of processors
MyRank: rank of the processor
Size: local size of the matrix
SizeGlo: global size of the matrix
Offset: offset of the global index
A[Size][SizeGlo]: local part of the matrix
X[Size]: local part of the solution vector
XOld[SizeGlo]: global solution vector
B[Size]: local part of the right-hand side vector
Error: local error
MaxError: global error
Epsilon: desired accuracy

repeat
   for i=0 to Size−1 do
      X[i] ← 0
      for j=0 to Offset−1 do
         X[i] ← X[i]+A[i][j]×XOld[j]
      end for
      for j=Offset+Size to SizeGlo−1 do
         X[i] ← X[i]+A[i][j]×XOld[j]
      end for
   end for
   for i=0 to Size−1 do
      X[i] ← (B[i]−X[i])/A[i][i+Offset]
   end for
   Error← 0
   for i=0 to Size−1 do
      Error ← max(Error, abs(X[i]−XOld[i+Offset]))
      XOld[i+Offset] ← X[i]
   end for
   for k=0 to NbProcs−1 do
      if k ≠ MyRank then
         Send(k, X)
      end if
   end for
   for k=0 to NbProcs−1 do
      if k ≠ MyRank then
         Recv(k, XOld[k×Size])
      end if
   end for
   AllReduce(Error, ErrorMax, MAX)
until stopping criteria is reached (MaxError ≤ Epsilon)
```

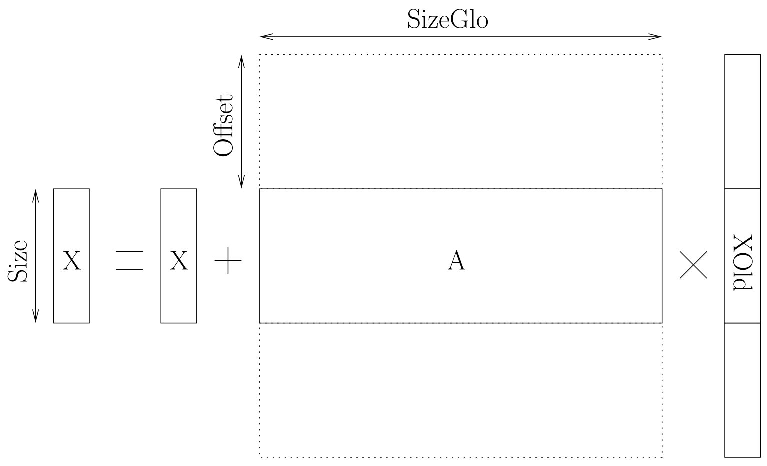

FIGURE 4.3: Splitting of the matrix for the synchronous Jacobi method.

and putting the result in the variable $ErrorMax$. In that case, the `MAX` operator is used in order to compute the maximum of the local error (or norm here). The result is available on each processor. As a consequence, the result is strictly identical to the result obtained using the sequential version (neglecting the potential rounding errors).

The communication part consists for each processor in sending its results (the vector X) to all the processors that need it. In the case of a dense matrix, processor k sends its result to all the other processors. Then a processor receives all the results and directly puts them in their right places in the array $XOld$; each place is computed in function of the rank of the sender and the (local) size of the matrix (considered constant). From a practical point of view, the programmer should rather use nonblocking communications in order to perform the exchange of data. According to the programming environment, it is possible that the use of blocking sends and receptions leads to a deadlock situation in which some processors are blocked while communicating simultaneously with each other.

With some high level implementation libraries (like MPI) it is possible to use a single call to a function to realize this exchange operation. In the code corresponding to this example we use it, so interested readers are invited to test it with MPI. For example, it is possible to use a procedure called *AllToAllV(X[Offset],X,Size)* that produces the same results as the communication part in Algorithm 4.3.

4.4.3 Synchronous conjugate gradient algorithm

The conjugate gradient algorithm can be parallelized using basically the same procedure, i.e., splitting the work and synchronizing every part of the code requiring it. Algorithm 4.4 describes the synchronous version of the conjugate gradient algorithm. In the following, we use *X[Offset]=Y* in order to copy the elements of the vector Y into X at the offset *Offset*. Likewise, for some operations we specify the number of elements that are concerned. For example, *P[Offset]=R (copy Size elements)* means that size elements of R are copied into P with the offset *Offset*.

Compared to the synchronous parallel Jacobi algorithm, the parallel version of the conjugate gradient requires more synchronizations. With the parallel Jacobi only two steps act as a synchronization, the exchange of data and the computation of the global error. In opposition, the parallel conjugate gradient algorithm contains twice the synchronization steps: three *AllReduce* operations and one *AllToAllV*.

4.4.4 Synchronous block Jacobi algorithm

The synchronous block Jacobi algorithm is relatively easy to write having the sequential version in mind. In fact, each processor is responsible for the computation of a block and after an iteration, all processors send their local solution to their neighbors that need it. In Algorithm 4.5, we describe the synchronous version of this algorithm. At the end of an iteration, processors exchange their local computation using an *AllToALLV* procedure. Then, they compute the global error. Consequently this algorithm requires two synchronization steps. In order to solve the local subsystem, each processor uses an appropriate method. In practice, depending on the size of the submatrix and its degree of density, a sparse or a dense direct method can be used. The solution of a subsystem is considered exact (neglecting rounding errors), so an iterative method is not considered for solving a local subsystem.

At each iteration, three main steps may be distinguished in the block Jacobi algorithm. The first one consists in updating the right-hand side using the dependencies of other processors. In Algorithm 4.5, this step updates the vector *BTmp*. The second step aims at solving the local subsystem on each processor. Using an existing solver obviously simplifies the programming of this method. According to the characteristics of the matrix, the choice of the inner solver may drastically change the efficiency of the parallel solver. Finally, the third step corresponds to the data exchanges and to the global error computation.

Algorithm 4.4 Synchronous conjugate gradient algorithm

```
NbProcs: number of processors
MyRank: rank of the processor
Size: local size of the matrix
SizeGlo: global size of the matrix
Offset: offset of the global index
A[Size][SizeGlo]: local part of the matrix
X[SizeGlo]: solution vector
R[Size]: local part of the residual vector
B[Size]: local part of the right-hand side vector
P[SizeGlo]: search direction vector
Q[Size]: local part of the orthogonal vector to the search direction
DotPQ, DotPQGlo: local and global scalar product of (P,Q)
Alpha, Beta, Rho, RhoGlo: scalar variables
Error: local error
MaxError: global error
Epsilon: desired accuracy

Offset ← Size×MyRank
R ← B−A×X
repeat
   Rho ← (R,R)
   AllReduce(Rho, RhoGlo, SUM)
   if i=1 then
      P[Offset] ← R (copy Size elements)
   else
      Beta ← RhoGlo/RhoOldGlo
      P[Offset] ← R+Beta×P[Offset] (copy Size elements)
   end if
   AllToAllV(P[Offset], P, Size)
   Q ← A×P
   DotPQ ← (P[Offset],Q)(for only Size elements)
   AllReduce(DotPQ, DotPQGlo, SUM)
   Alpha ← RhoGlo/DotPQGlo
   X[Offset] ← X+Alpha×P[Offset] (for Size elements)
   R ← R−Alpha×Q
   RhoOldGlo ← RhoGlo
   Error ← 0
   for i=0 to Size−1 do
      Error ← max(Error, abs(R[i]))
   end for
   AllReduce(Error, ErrorMax, MAX)
until stopping criteria is reached (MaxError ≤ Epsilon)
```

Algorithm 4.5 Synchronous block Jacobi algorithm

```
NbProcs: number of processors
MyRank: rank of the processor
Size: local size of the matrix
SizeGlo: global size of the matrix
Offset: offset of the global index
A[Size][SizeGlo]: local part of the matrix
X[Size]: local part of the solution vector
B[Size]: local part of the right-hand side vector
BTmp[Size]: intermediate local part of the right-hand side vector
XOld[SizeGlo]: global solution vector
Error: local error
MaxError: global error
Epsilon: desired accuracy

Offset← Size×MyRank
repeat
   for i=0 to Size−1 do
      BTmp[i]← B[i]
   end for
   for i=0 to Size−1 do
      for j=0 to Offset−1 do
         BTmp[i]← BTmp[i]−A[i][j]×XOld[j]
      end for
      for j=Offset+Size to SizeGlo do
         BTmp[i]← BTmp[i]−A[i][j]×XOld[j]
      end for
   end for
   X← Solve(A, BTmp)
   Error← 0
   for i=0 to Size−1 do
      Error ← max(Error, abs(X[i]−XOld[i+Offset]))
      XOld[i+Offset]← X[i]
   end for
   AllToAllV(XOld[Offset], XOld, Size)
   AllReduce(Error, ErrorMax, MAX)
until stopping criteria is reached (MaxError ≤ Epsilon)
```

4.4.5 Synchronous multisplitting algorithm for solving linear systems

This algorithm comes directly from the formulation of Equations (4.3)-(4.5). In the following we explain how to obtain the algorithm without overlapping components and then we introduce the overlapping of components.

The first step consists in defining the weighting matrices. Without introducing overlapping, those matrices are diagonal and either contain 1 or 0 on the diagonal. For example, if we take three processors ($L = 3$), the weighting matrices are defined as in Figure 4.4.

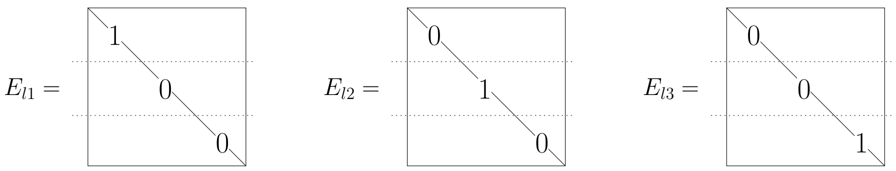

FIGURE 4.4: An example with three weighting matrices.

Having defined those matrices, we need to define the matrices M_l and N_l. With Equation (4.4) and by defining $T^{(l)}$ as in Equation (4.6), with three processors, we obtain the following system

$$\begin{cases} y^1 = M_1^{-1}N_1(E_{11}x^1 + E_{12}x^2 + E_{13}x^3) + M_1^{-1}b \\ y^2 = M_2^{-1}N_2(E_{21}x^1 + E_{22}x^2 + E_{23}x^3) + M_2^{-1}b \\ y^3 = M_3^{-1}N_3(E_{31}x^1 + E_{32}x^2 + E_{33}x^3) + M_3^{-1}b \end{cases} \tag{4.23}$$

In that system, matrices M_l^{-1} and N_l and vectors y^l, x^l and b are not decomposed. From a practical point of view, a processor does not handle the whole vectors and matrices, it only has the parts it is in charge of. In the example with three processors, each processor has a third of data. So we can define y'^l, x'^l and b'^l that correspond to the parts handled by the processors. And Equation (4.23) can be rewritten as

$$\begin{cases} y'^1 = (A_{11}^{-1} \times -A_{12})x'^2 + (A_{11}^{-1} \times -A_{13})x'^3 + A_{11}^{-1}b'^1 \\ y'^2 = (A_{22}^{-1} \times -A_{21})x'^1 + (A_{22}^{-1} \times -A_{23})x'^3 + A_{22}^{-1}b'^2 \\ y'^3 = (A_{33}^{-1} \times -A_{31})x'^1 + (A_{33}^{-1} \times -A_{32})x'^2 + A_{33}^{-1}b'^3 \end{cases} \tag{4.24}$$

with, for example, the splittings depicted in Figure 4.5.

By multiplying each previous equation by A_{ii} we obtain:

$$\begin{cases} A_{11} \times y'^1 = b'^1 - A_{12}x'^2 - A_{13}x'^3 \\ A_{22} \times y'^2 = b'^2 - A_{21}x'^1 - A_{23}x'^3 \\ A_{33} \times y'^3 = b'^3 - A_{31}x'^1 - A_{32}x'^2 \end{cases} \tag{4.25}$$

$$A = \begin{array}{|c|c|c|}\hline A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \\ \hline \end{array}$$

$$M_1 = \begin{array}{|c|c|c|}\hline A_{11} & 0 & 0 \\ \hline 0 & A_{22} & A_{23} \\ \hline 0 & A_{32} & A_{33} \\ \hline \end{array} \qquad N_1 = \begin{array}{|c|c|c|}\hline 0 & A_{12} & A_{13} \\ \hline A_{21} & 0 & 0 \\ \hline A_{31} & 0 & 0 \\ \hline \end{array}$$

$$M_2 = \begin{array}{|c|c|c|}\hline A_{11} & 0 & 0 \\ \hline 0 & A_{22} & 0 \\ \hline 0 & 0 & A_{33} \\ \hline \end{array} \qquad N_2 = \begin{array}{|c|c|c|}\hline 0 & A_{12} & A_{13} \\ \hline A_{21} & 0 & A_{23} \\ \hline A_{31} & A_{32} & 0 \\ \hline \end{array}$$

$$M_3 = \begin{array}{|c|c|c|}\hline A_{11} & A_{12} & 0 \\ \hline A_{21} & A_{22} & 0 \\ \hline 0 & 0 & A_{33} \\ \hline \end{array} \qquad N_3 = \begin{array}{|c|c|c|}\hline 0 & 0 & A_{13} \\ \hline 0 & 0 & A_{23} \\ \hline A_{31} & A_{32} & 0 \\ \hline \end{array}$$

FIGURE 4.5: An example of possible splittings with three processors.

In this example, each processor i has a local linear subsystem to solve. The right-hand side contains the corresponding part of the vector b'^{i} minus all the block off-diagonal (A_{ij} with $j \neq i$) multiplied by the corresponding x'^{j} with $j \neq i$.

From a practical point of view, we call the off-diagonal block the dependencies of the computation. So, processor 1 depends, respectively, on processors 2 and 3 if blocks A_{12} and A_{13} are, respectively, nonempty. In the following practical algorithm we want to gather all dependencies before the current processor in an array called *LeftDep* and all the dependencies after the current processor in an array called *RightDep*. Likewise, each processor has two vectors $XLeft$ and $XRight$.

In Figure 4.6, the decomposition of the matrix is illustrated. Each processor is in charge of a rectangular part of the matrix. This rectangular part is split into three parts. The left dependencies (DepLeft) involve components computed by processors whose rank is strictly smaller than the one of the considered processor. The submatrix (noted A) is the square matrix that a processor is in charge of; it corresponds to the matrix A_{ii} in Equation (4.25). And finally, the right dependencies (DepRight) involve components computed by processors whose rank is strictly greater than the one of the considered processor. With such a decomposition, a processor needs to solve:

$$A \times X = B - DepLeft \times XLeft - DepRight \times XRight \qquad (4.26)$$

which exactly corresponds to the lines in Equation (4.25). As soon as it has computed the solution of the subsystem, this solution needs to be sent to all processors depending on it.

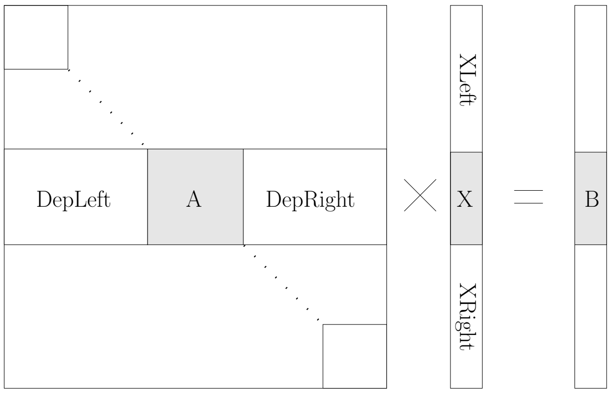

FIGURE 4.6: Decomposition of the matrix.

Algorithm 4.6 illustrates the synchronous version of the multisplitting algorithm to solve linear systems. At the beginning of an iteration, a processor computes the right-hand side as in Equation (4.26). Then it solves the linear system composed of the submatrix it is in charge of with the right-hand side that takes into account the dependencies just computed. To solve this linear system, it is possible to use either a direct algorithm or an iterative one (with or without using a preconditioner); in this latter case we obtain a two-stage method. In Section 6.4, we report an experiment that highlights the impact of using this or that algorithm. The next step consists in exchanging dependencies. Before starting the iterative process, processors exchange their dependencies in order to initialize the arrays *DependsOnMe* and *IDependOn*. This part involves the use of the offset computed on each processor. Moreover, a processor takes into consideration what parts of the solution vector it needs. With those arrays, a processor knows its neighbors. Consequently, it can send to each of its neighbors the part of its vector that they need. Then it is ready to receive dependencies from its neighbors. According to the rank of a neighbor, a processor integrates the dependency in *DepLeft* or *DepRight*. Globally, this exchange part acts as a synchronization step. The last step lies in computing the error locally and then globally using for a second time a synchronization step so that all the processors know the global error.

As explained, one particularity of the multisplitting method is that it allows the processors to overlap components in order to speed up the convergence. The principle is to let some processors compute simultaneously some com-

Algorithm 4.6 Synchronous linear multisplitting algorithm

```
NbProcs: number of processors
MyRank: rank of the processor
Size: local size of the matrix
SizeGlo: global size of the matrix
Offset: offset of the global index
A[Size][Size]: local block-diagonal part of the matrix
DepLeft[Size][Offset]: submatrix with left dependencies
DepRight[Size][SizeGlo-Offset-Size]: submatrix with right dependencies
DependsOnMe[NbProcs]: array of the dependent processors
IDependOn[NbProcs]: array of the processors this processor depends on
B[Size]: right-hand side vector of the subsystem
X[Size], XOld[Size]: local part solution vectors of the subsystem
XLeft[Offset]: left part of the solution vector of the system
XRight[SizeGlo-Offset-Size]: right part of the solution vector of the system
BLoc[Size]: array containing the local computations on the right-hand side
TLoc[Size]: array used for the receptions of the dependencies
Error: local error
MaxError: global error
Epsilon: desired accuracy

repeat
   BLoc ← B
   if MyRank≠0 then
      BLoc ← BLoc−DepLeft×XLeft
   end if
   if MyRank ≠ NbProcs−1 then
      BLoc ← BLoc−DepRight×XRight
   end if
   X ← Solve(A, BLoc)
   for i=0 to NbProcs−1 do
      if i ≠ MyRank and DependsOnMe[i] then
         Send(i, PartOf(X, i))
      end if
   end for
   for i=0 to NbProcs−1 do
      if i ≠ MyRank and IDependOn[i] then
         Recv(i, TLoc)
         Update XLeft or Xright with TLoc according to the processor $i$
      end if
   end for
   Error← 0
   for i=0 to Size−1 do
      Error ← max(Error, abs(X[i]−XOld[i]))
      XOld[i]← X[i]
   end for
   AllReduce(Error, ErrorMax, MAX)
until stopping criteria is reached (MaxError ≤ Epsilon)
```

ponents and to mix the results in order to obtain an accurate result faster. It corresponds to splitting the matrix into rectangular matrices that are not disjoint (J sets in the theoretical framework). In Figure 4.7, we give an example with a small matrix of size 9×9 for which the corresponding linear system is solved with three processors using one overlapped component with each neighbor. Without using overlapping, each processor has three components (processor 1 has components 1 to 3, processor 2 has components 4 to 6 and processor 3 has components 7 to 9). If we allow some components to be overlapped, processors with only one neighbor (i.e., processors 1 and 3 in the figure) have four components. Processor 2 has five components. In Figure 4.7, the hatched parts represent the submatrices that processors are in charge of (matrix A in Algorithm 4.6) and black dots represent non-null values of the matrix. Parts in the matrix that are doubly hatched highlight components that are computed by two processors. Subvectors x'^i (X in Algorithm 4.6) also have overlapped components that are represented in gray in the figure. Non-null values that are not in submatrices represent dependencies. In Figure 4.7, circled dots illustrate dependencies that are simultaneously computed by two processors. Dependencies on lines 1 and 2 are computed by processors 2 and 3. Likewise, dependencies on lines 8 and 9 are computed by processors 1 and 2.

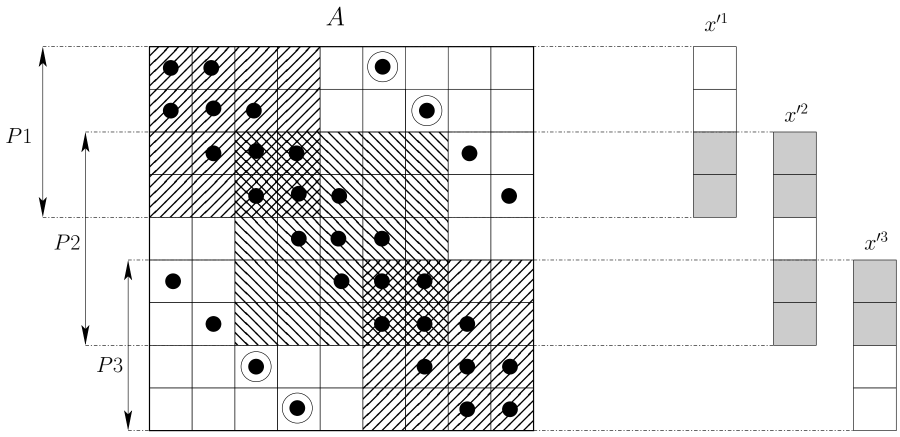

FIGURE 4.7: An example of decomposition of a 9×9 matrix with three processors and one component overlapped at each boundary on each processor.

There are multiple ways to mix the overlapped components. In the following, we explain four ways to mix overlapped components.

4.4.5.1 Overlapping strategy that uses locally computed values

In this case, a processor only uses components that it has computed while ignoring the values corresponding to the same set of components computed by its neighbors. With the example defined in Figure 4.7, it consists in defining the weighting matrices as in Figure 4.8. In this figure, matrices E_{1k}, E_{2k} and E_{3k}, respectively, correspond to weighting matrices of processors 1, 2 and 3. Overlapped components are represented in gray in the figure. For example, components 3 and 4 are computed simultaneously by processors 1 and 2. For a processor i, we can remark that the sum of weighted matrices E_{ik} is equal to the identity matrix (as expressed in Equation (4.5)). Hence, with this strategy, processor 1 uses its components 1 to 4, it uses components 5 and 6 of processor 2 and components 7 to 9 of processor 3. Processor 2 uses its components 3 to 7 and uses components 1 and 2 of processor 1 and components 8 and 9 of processor 3. Processor 3 proceeds similarly to processor 1; since it has its 4 components, it uses 2 components of processor 2 and 3 components of processor 1. This strategy is quite easy to implement since it does not require any mixing of overlapped components. It simply consists in using all the components computed by a processor.

4.4.5.2 Overlapping strategy that uses values computed by close neighbors

This strategy has similarities to the previous one because it does not require any mixing of overlapped components either. The principle consists in using all the overlapped components of its close neighbors. Weighting matrices, corresponding to the same example, are illustrated in Figure 4.9. In that case, processor 1 is close to processor 2 but not to processor 3. So processor 1 only uses its components 1 and 2, it uses components 3 to 6 that are computed by processor 2 and it uses components 7 to 9 computed by processor 3. Processor 2 has two close neighbors (processors 1 and 3); it uses components 1 to 4 of processor 1, it uses its single component 5 and it uses components 6 to 9 of processor 3. Processor 3 proceeds similarly to processor 1 and it uses components 1 to 3 of processor 1, components 4 to 7 of processor 2 and its components 8 and 9. Compared to the previous strategy, this one requires more data exchange since a processor requires all overlapped components of its close neighbors.

4.4.5.3 Overlapping strategy that mixes overlapped components with close neighbors

With this strategy a processor mixes its overlapped components with its close neighbors. In Figure 4.10, we give an example of the mixing which consists in taking half of the value computed by a processor and half of the value computed by the close neighbor. So, processor 1 has its components 1 and 2 and it mixes its components 3 and 4 with processor 2, it uses components

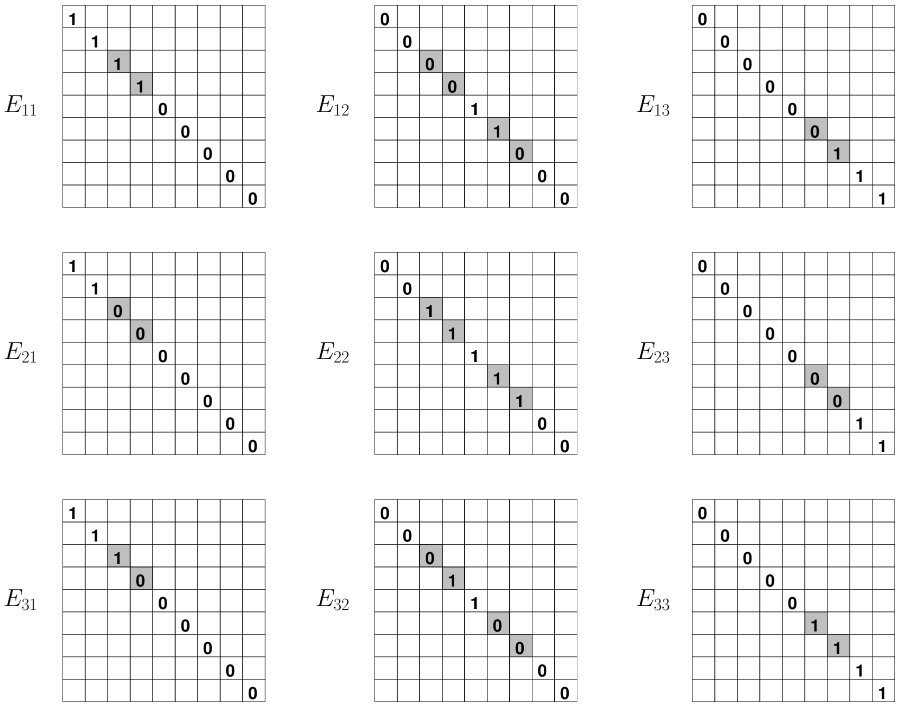

FIGURE 4.8: Overlapping strategy that uses values computed locally.

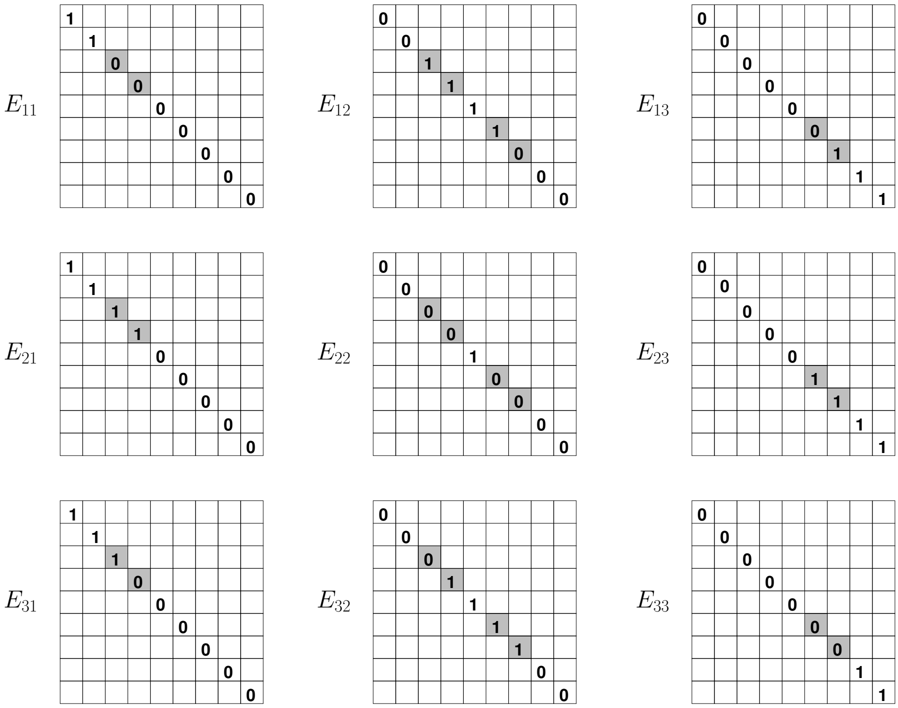

FIGURE 4.9: Overlapping strategy that uses values computed by close neighbors.

5 and 6 of processor 2 and components 7 to 9 of processor 3. Processor 2 uses components 1 and 2 of processor 1, it mixes its components 3 and 4 with processor 1, it uses it component 5, it mixes its components 6 and 7 with processor 3 and it uses components 8 and 9 of processor 3. Processor 3 uses components 1 to 3 of processor 1, it uses components 4 and 5 of processor 2, it mixes its components 6 and 7 with processor 2 and it uses its components 8 and 9. Compared to the previous strategy, the amount of data exchange is strictly equal. With this strategy it is possible to use different ratios of mixing as soon as the sum of ratios on one line of all the matrices E_{lk} is equal to 1. For instance, it is possible to take 75% of the computed components and 25% of the values computed by the other processor.

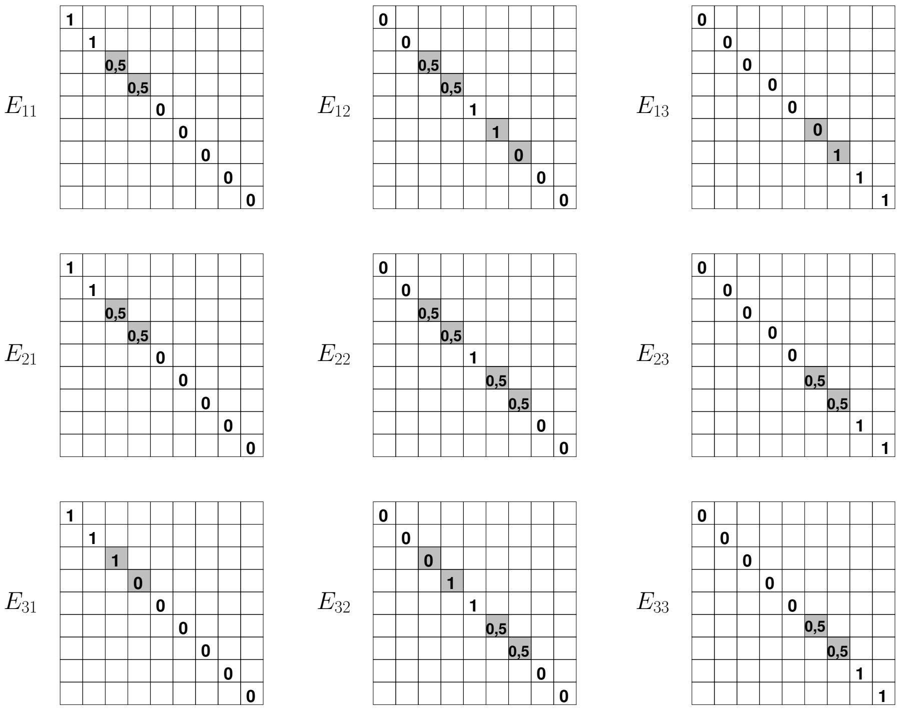

FIGURE 4.10: Overlapping strategy that mixes overlapped components with close neighbors.

4.4.5.4 Overlapping strategy that mixes all overlapped components

This strategy mixes all components that are overlapped (not only with its close neighbors). In Figure 4.11 we illustrate a possible example of values of the weighted matrices E_{lk} for this strategy. All the gray parts, corresponding to overlapped components, contain values that are different from 0 and 1. This strategy offers the most freedom to mix overlapping components. In our example, all processors use components 1 and 2 from processor 1, they mix components 3 and 4 from processors 1 and 2, they use component 5 of processor 2, they mix components 6 and 7 from processors 2 and 3 and they use components 8 and 9 of processor 3. If different values of mixing are used according to processors, all overlapped components must be sent to processors that need them. Consequently the amount of data transfered may be greater.

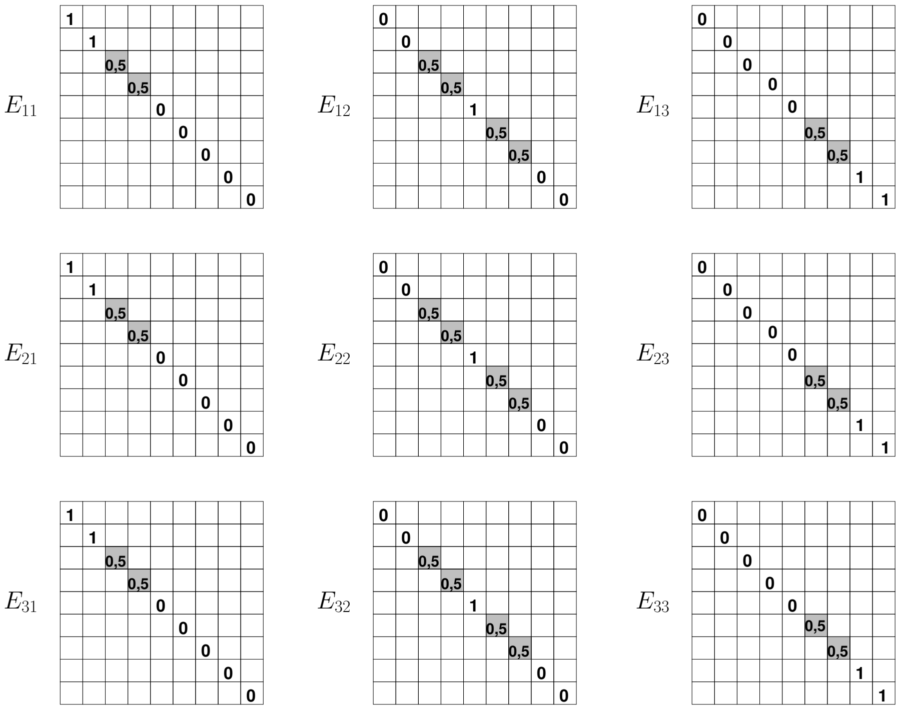

FIGURE 4.11: Overlapping strategy that mixes all overlapped values.

Implementing all those strategies in Algorithm 4.6 is quite easy. The matrix distribution should be achieved by taking into account the overlapped components on each processor. So, the offsets and the local size of matrices are different. Then, according to the overlapping strategy chosen, the amount of data might be different. For example with the first strategy presented, the amount of data is smaller, whereas with the last strategy the amount of data transfered is greater. According to the structure and dependencies of the problem, the choice of the best strategy is not a trivial task.

4.4.6 Synchronous Newton-multisplitting algorithm

In the previous sections we have explained how to implement the multisplitting method in order to solve linear systems. Now we describe how to design the algorithm based on the Newton-multisplitting method. As described formerly, this method uses the Newton method to linearize the problem. Using synchronous iterations, the algorithm first builds the Jacobian matrix and then it needs to solve the linear system obtained. The multisplitting method to solve a linear system is used for this. In Algorithms 4.7 and 4.8, both methods are coupled. The outer iterations perform the Newton iteration, whereas the inner ones solve the linear system. Variables of this algorithm are described in Algorithm 4.7 and its core is given in Algorithm 4.8.

At each Newton iteration, a processor starts by computing the rectangular part of the Jacobian matrix it is in charge of. Then it computes the right-hand side of the nonlinear function as described in Equation (4.16). Of course, each processor has a different set of components according to the block distribution. As soon as the Jacobian submatrices have been defined simultaneously on processors, they start to solve the linear system using the multisplitting algorithm for linear systems. To solve the subsystem it is possible to use a direct method or an iterative one. In this latter case, we obtain a two-stage algorithm. The Jacobian is computed using the same computation as in the sequential algorithm. The only difference is that the Jacobian is distributed on all processors, as described in Figure 4.12. The computation of $-F$ is also distributed across the processors.

After the computation of the Jacobian, the method consists in solving the whole linear system using the multisplitting method for linear systems. So, until global convergence of the multisplitting algorithm, a processor computes the local right-hand side and updates it using the dependencies computed by its neighbors. In the algorithm, J represents the local submatrix for a processor. $JDepLeft$ and $JDepRight$, respectively, correspond to $DepLeft$ and $DepRight$ defined in the multisplitting algorithm for linear systems, described previously. At each multisplitting iteration, a processor solves the subsystem composed of the Jacobian submatrix and the right-hand side using a sequential solver. Then, it sends its DX vector part to each of its neighbors that need it, and it receives the part of the solution of its neighbors and updates the vectors $DXLeft$ and $DXRight$. Finally, it computes the local error of the multisplitting process. When the multisplitting method has globally converged, a processor computes the local error of the Newton process. Pro-

Algorithm 4.7 Variables used in the synchronous Newton-multisplitting algorithm

```
NbProcs: number of processors
MyRank: rank of the processor
Size: local size of the matrix
SizeGlo: global size of the matrix
Offset: offset of the global index
J[Size][Size]: local block-diagonal part of the Jacobian matrix
JDepLeft[Size][Offset]: submatrix with left dependencies of the Jacobian
JDepRight[Size][SizeGlo-Offset-Size]: submatrix with right dependencies of
the Jacobian
DependsOnMe[NbProcs]: array of the dependent processors
IDependOn[NbProcs]: array of the processors this processor depends on
F[Size], FLoc[Size]: right-hand side vectors of the subsystem
X[SizeGlo]: solution vector of the Newton subsystem
DX[Size], DXOld[Size]: local part solution vectors of the multisplitting sub-
system
DXLeft[Offset]: left part of the solution vector of the system
DXRight[SizeGlo-Offset-Size]: right part of the solution vector of the system
TLoc[Size]: array used for the receptions of the dependencies
ErrorNewton: local error of the Newton process
MaxErrorNewton: global error of the Newton process
ErrorMulti: local error of the multisplitting
MaxErrorMulti: global error of the multisplitting process
EpsilonMulti: desired accuracy for the multisplitting process
EpsilonNewton: desired accuracy for the Newton process
```

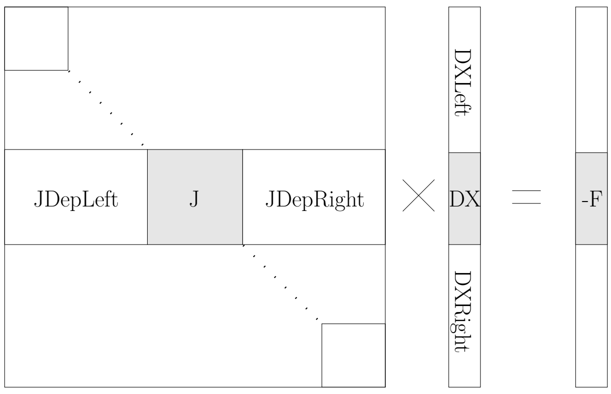

FIGURE 4.12: Decomposition of the Newton-multisplitting.

Algorithm 4.8 Synchronous Newton-multisplitting algorithm

```
repeat
  if first iteration or required then
    Computation of the Jacobian rectangular matrix and storage of the
    respective parts into $J$, $JDepLeft$ and $JDepRight$
  end if
  Computation of −F depending on X from components Offset to
  Offset+size−1 and storage of the result into F
  repeat
    FLoc ← F
    if MyRank ≠ 0 then
      FLoc ← FLoc−JDepLeft×DXLeft
    end if
    if MyRank ≠ NbProcs−1 then
      FLoc ← FLoc−JDepRight×DXRight
    end if
    DX ← Solve(J, FLoc)
    for i=0 to NbProcs−1 do
      if i ≠ MyRank and DependsOnMe[i] then
        Send(i, PartOf(DX, i))
      end if
    end for
    for i=0 to NbProcs−1 do
      if i ≠ MyRank and IDependOn[i] then
        Recv(i, TLoc)
        Update DXLeft or DXRight with TLoc according to processor $i$
      end if
    end for
    ErrorMulti← 0
    for i=0 to Size−1 do
      ErrorMulti ← max(ErrorMulti, abs(DX[i]−DXOld[i]))
      DXOld[i]← DX[i]
    end for
    AllReduce(ErrorMulti, MaxErrorMulti, MAX)
  until stopping criteria of multisplitting is reached
        (MaxErrorMulti ≤ EpsilonMulti)
  ErrorNewton← 0
  for i=0 to Size−1 do
    X[Offset+i] ← X[Offset+i]+DX[i]
    ErrorNewton ← max(ErrorNewton, abs(DX[i]))
  end for
  AllToAllV(X[Offset], X, Size)
  AllReduce(ErrorNewton, MaxErrorNewton, MAX)
until stopping criteria of Newton is reached
      (MaxErrorNewton ≤ EpsilonNewton)
```

cessors execute Newton iterations until the global error is lower than a given threshold.

Analyzing the number of synchronizations of this algorithm we can remark that there are as many synchronizations as multisplitting iterations and Newton iterations (considering that there is only one synchronization per iteration, we have seen previously that this is not the case). Thus this algorithm requires quite an important number of synchronizations.

4.5 Convergence detection

We discuss here the problem of convergence detection in iterative processes of the form:

$$x^{(k+1)} = G(x^{(k)}) \tag{4.27}$$

where $x^{(k+1)}$ and $x^{(k)}$ are the global state vectors at the respective iterations $k+1$ and k, and G is a contraction. The useful property of contractions is that their convergence is ensured. This is why most of the iterative algorithms currently in use are contractions. In fact, an important constraint when designing an iterative method is precisely to ensure that it is a contraction.

However, most of the theoretical results related to the convergence of iterative processes, including contractions, are of limited interest in practice since they are often based on properties which are not directly calculable or whose computation cost is of the same order as the problem to solve. Hence, as explained in [33], practical methods for proving the convergence of iterative methods generally consist in finding a suitable norm for which it can be shown that each iteration reduces the distance between the current global state vector and the fixed point which represents the solution of the problem.

Unfortunately, it is possible to ensure that an iterative process is a contraction without being able to find the suitable norm which allows us to detect its convergence in practice. For example, in linear problems, we know that we have a contraction when the spectral radius of the iteration matrix is smaller than one but there is no information about the norm to use in practice. Moreover, as already mentioned in Section 1.2, a contraction is norm dependent and may be contractive with a given norm and non-contractive with another one. This is an important problem which may induce some difficulties in the convergence detection.

Indeed, when the contraction norm is known (let's note it $||\ .\ ||_{\mathcal{C}}$) there is no problem detecting the convergence since, by definition, we have:

$$\forall x, y, \quad ||G(x) - G(y)||_{\mathcal{C}} \leq L||x - y||_{\mathcal{C}} \quad \text{with} \quad L < 1 \tag{4.28}$$

which implies

$$||x^{(k+1)} - x^{(k)}||_{\mathcal{C}} < ||x^{(k)} - x^{(k-1)}||_{\mathcal{C}} \tag{4.29}$$

So, the distance between two global state vectors obtained from two consecutive iterations decreases according to the contraction norm $|| \, . \, ||_{\mathcal{C}}$. That distance between two consecutive iterations is often called the *residual* and, in some ways, represents the progression speed of the iterative process. Thus, when the right norm is used, the residual monotonously decreases toward zero, without reaching it if the convergence is asymptotic. However, when the residual becomes small enough, it can be assumed that the iterative process is sufficiently close to the exact solution to detect the convergence and stop the process. Hence, the residual is regularly compared to a given threshold defining a sufficiently small progression speed of the process to assume its stabilization. A schematic example of such a behavior is given in Figure 4.13.

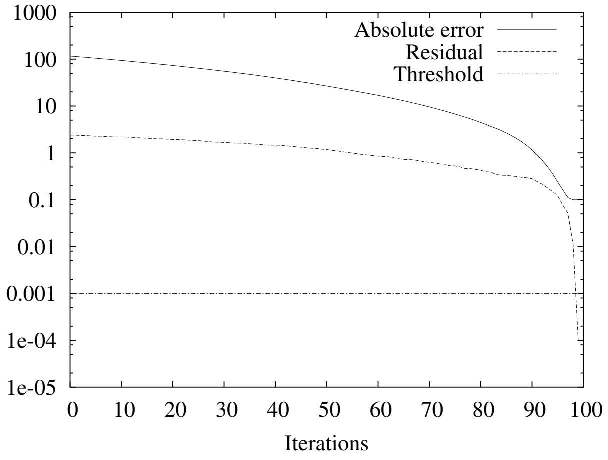

FIGURE 4.13: Monotonous residual decreases toward the stabilization according to the contraction norm.

Nevertheless, as mentioned above, it is not always possible to know the contraction norm and it is then necessary to arbitrarily choose one metric among all the possible ones. The most common metrics are the Euclidean norm:

$$||x^{(k)} - x^{(k-1)}||_2 = \sqrt{\sum_{i=1}^{n} (x_i^{(k)} - x_i^{(k-1)})^2} \tag{4.30}$$

and the max norm:

$$||x^{(k)} - x^{(k-1)}||_\infty = \max_i |x_i^{(k)} - x_i^{(k-1)}| \tag{4.31}$$

where $x_i^{(k)}$ and $x_i^{(k-1)}$ are the respective i^{th} component of state vectors $x^{(k)}$ and $x^{(k-1)}$.

So, when the norm used is not the contraction norm, nothing ensures that Equations (4.28) and (4.29) still hold. Using such an arbitrary norm makes the convergence detection far more difficult as there is no more valuable information about the position of the current state according to the exact solution. Thus, even when the residual becomes very small, nothing ensures us that the process is actually close to the exact solution. Typically, if the path followed by the iterative process toward the solution in the state space includes smaller variations than the chosen threshold according to the chosen norm, the convergence may be detected even though the current state may still be far from the solution. Moreover, even when the iterative process has a monotonous evolution toward the solution, i.e., when the distance between the current state vector and the exact solution always decreases from an iteration to the following one, the residual may not be monotonous. A schematic illustration of such a case is depicted in Figure 4.14.

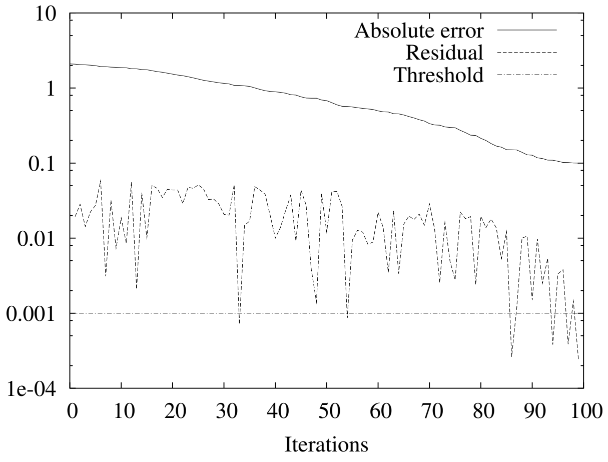

FIGURE 4.14: A monotonous error evolution and its corresponding non-monotonous residual evolution.

As can be seen, the problem induced by such variations of the residual is that important slow-downs like the one at iteration 33 may not correspond to the final stabilization of the process and lead to a false convergence detection if the threshold is set too high. On the other hand, if the threshold is set too low, the iterative process may not converge in reasonable time when the convergence is asymptotic. Some attempts have been made to overcome that problem by taking into account several consecutive residuals, for example:

$$residual^{(k)} = ||x^{(k)} - x^{(k-1)}|| + ||x^{(k-1)} - x^{(k-2)}|| \qquad (4.32)$$

in order to avoid false detections due to sharp slow-downs. However, that does not completely solve the problem since temporary slow-downs under the given threshold may have arbitrary lengths.

Thus, the choice of the norm used to compute the residual is a critical point in the design of iterative algorithms and the setting of its associated threshold often requires a careful analysis of the treated problem in order to ensure an appropriate convergence detection.

4.6 Exercises

1. Show that a nonsingular M-matrix has the form
$$sI - B,$$
where $B \geq 0$ and $s > \rho(B)$.

2. Give examples of M-matrices and compute their principal minors.

3. Let A be a square matrix with $A_{i,j} \leq 0$ for $i \neq j$. Show that A is an M-matrix if and only if
$$A + \varepsilon I$$
is a nonsingular M-matrix for all $\varepsilon > 0$.

4. Consider the two-point boundary-value problem:
$$-\frac{d^2u}{dx^2} = 4\pi^2 \sin 2\pi x,\ 0 \leq x \leq 1 \tag{4.33}$$
and
$$u(0) = u(1) = 0.$$
   (a) Use the second central difference formulae with a constant step size $h = 1/(n+1)$ to approximate $\frac{d^2u}{dx^2}$ and show that the discrete approximation is the solution of a linear system $Au = b$ where $u = (u_1, ..., u_n)$.
   (b) For $n = 3$ show that
$$A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix},\ b = \frac{\pi^2}{4}\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}.$$
   (c) Use a direct method to solve the u_i.
   (d) Use the Jacobi and the Gauss-Seidel methods to find the discrete approximation.

(e) Compare the results to the true solution $u = \sin 2\pi x$ at $x = 1/4$, $x = 1/2$, $x = 3/4$ (Berman and Plemmons [31]).

5. Write a program to solve the two-point boundary-value problem (4.33) using Jacobi algorithm, SOR algorithm with optimum relaxation parameter for $n = 50$, $n = 200$, $n = 500$. Discuss the obtained results (Berman and Plemmons [31]).

6. Show that the system
$$\begin{cases} e^x - y = 0 \\ x - e^{-y} = 0 \end{cases}$$
has only one solution and write a program to solve it by the Newton method.

7. Write a program to solve the following system of nonlinear equations by the Newton method:
$$\begin{cases} x^2 + y^2 + z^2 - 3 &= 0 \\ xy + xz - 3yz + 1 &= 0 \\ x^2 + y^2 - z^2 - 1 &= 0. \end{cases}$$
Study the convergence of the Newton method.

8. Consider the Laplace equation
$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0, \ \ (x, y) \in [0, 2] \times [0, 1],$$
with
$$\begin{aligned} &f(0, y) = 0, \ f(2, y) = 6 \\ &f(x, 0) = 6, \ \forall x \in [1, 2] \\ &\frac{\partial f}{\partial y}(x \leq 1, y = 0) = 0, \ \frac{\partial f}{\partial y}(x, y = 1) = 0. \end{aligned}$$

(a) By using the finite difference method to approximate the second derivatives and following the illustration example of Chapter 1, write the linear system $Au = b$ whose solution coincides with the approximate solution of the above Laplace equation.

(b) By choosing step sizes $\Delta x = 2 \times 10^{-4}$ and $\Delta x = 10^{-4}$, write a program to solve the obtained linear system by the Jacobi algorithm and the Gauss-Seidel algorithm.

(c) Propose and write a program to solve the linear system by a multisplitting algorithm on 10 processors.

(d) Propose and write a program to solve the linear system by a two-stage multisplitting algorithm on 10 processors.

9. Consider the Poisson equation

$$-\Delta u = -32x(1-x)y(1-y) \text{ on } \Omega =]0,1[^2,$$
$$u = 0 \text{ on } \partial\Omega =]0,1[\times \{0,1\} \bigcup \{0,1\} \times]0,1[.$$

   (a) Following the above exercise, write a program to solve in parallel the discretized solution of this Poisson equation by the conjugate gradient method ($\Delta x = \Delta y = 10^{-4}$).

   (b) Propose and write a program to solve the linear system by different multisplitting algorithms on 10 processors.

   (c) Compare the overall times of the synchronous executions obtained with the different algorithms but with the same precision.

10. In all the algorithms presented in the implementation section, we have used the AllToAllV procedure that allows all processors to broadcast a part of a vector that they have computed. Write this procedure using only Send and Recv operations.

11. Implement all the algorithms presented in the implementation section using blocking and nonblocking receptions. Try to measure the performances with twenty processors or so.

12. Using an AllReduce operation allows us to simply diffuse the maximum of the local convergence on all processors. Try to implement the same thing using a master processor that will receive the local convergence of all processors, compute the global value and then diffuse the result to all processors.

13. With a sparse matrix, split into rectangular matrices as in Figure 4.6 on each processor, implement an algorithm that allows us to compute the arrays *DependsOnMe* and *IDependOn* as in Algorithm 4.6.

14. Most systems needing to be solved are sparse. Implement algorithms described in this chapter with a sparse matrix representation. Then compare the behavior of an algorithm optimized with a naive implementation.