Chapter 5

Asynchronous Iterations

Introduction

In the grid computing framework, especially when the clusters are distant, the ratio *computation time/communication time* can be weak and thus give a considerable importance to the communications. For this reason, powerful algorithms, such as those based on the minimization of a function, can paradoxically become less powerful in such environments. The synchronizations between the iterates provide the same convergence of algorithms as in the sequential case. Nevertheless, those synchronizations are penalizing in the distant clusters framework.

The asynchronous algorithms allow processors to compute at their own rhythms and to send the data when they become available. The communications as well as the iterations are desynchronized, avoiding the penalizing synchronizations and carrying out a kind of automatic overlapping of communications by computations. It is, however, necessary to precede any implementation of asynchronous iterative algorithms by a study of their convergence; this is due to the desynchronization of the iterations (notice that the study of the convergence is also necessary for all the iterative algorithms, even in the synchronous or the sequential framework).

In this chapter, we are interested in the multisplitting methods and their *two-stage* variants and in their coupling with the Newton method. Multisplitting algorithms allow us to carry out coarse grained parallelism which is very suited in the field of grid computing. Moreover, the two-stage multisplitting algorithms make it possible to choose, at the level of each processor, the best adapted sequential algorithm to the subproblem. We thus obtain coarse grained asynchronous algorithms with a coupling of different sequential algorithms.

5.1 Advantages of asynchronous algorithms

Contrary to synchronous implementations, in Asynchronous Iterations - Asynchronous Communication (AIAC) execution modes the processors are not coordinated in order to obtain a solution of a fixed point problem. Some processors are allowed to compute faster than others; some communications are allowed to be more frequent than others. The delays between processors are unpredictable and the transmission of messages may be accomplished in an unspecified order. Asynchronous iterations have been introduced in [38] by Chazan and Miranker for linear problems under the name *chaotic relaxation*. The pioneers in the study and the generalization to asynchronous algorithms are Miellou [86], Baudet [30], Robert, Charnay and Musy [101], Bertsekas and Tsitsiklis [32, 33], Bahi et al. [25] and Bahi [12].

The asynchronous iterations model describes a wide generalization of the successive approximation method in the case of a fixed point mapping defined on a product space, or even a product set.

This formulation is sufficiently general in order to contain:

- The successive approximation method which includes an inherent parallelism

- The Gauss-Seidel method which is often strictly sequential. The first of these two standard algorithms is well designed for parallelization while the second one is often, but not always, sequential. On the other hand, Gauss-Seidel iterations satisfy the so-called *Gauss principle*, which asserts that a new partial result is immediately used anew. Jacobi iterations do not satisfy the Gauss principle. The aim of asynchronous iterations is to satisfy in the best way the Gauss principle in a parallel framework. In some sense they afford a compromise between the usual good properties of the Jacobi and Gauss-Seidel methods.

Asynchronous executions have several potential advantages; we list some of them below (see [80]):

1. They reduce the effect of bottlenecks. Indeed, if for example the communication link between two processors is drastically slowed down then, contrary to synchronous executions, all the processors will go on and the two processors with the slow link will not slow down the processors which do not directly depend on them.

2. They reduce the synchronization penalty. A processor can compute the next iteration without waiting for the iterations computed by slower processors.

3. They are well designed for systems in which synchronizations are unrealistic such as very dense systems and for systems where global information is impossible to obtain such as decentralized systems.

4. They are easily restartable. For example, suppose that while solving an optimization problem, a change happens in one parameter, as may be the case in data networks. Then, while in synchronous computations the system has to be stopped and restarted, in asynchronous executions, the parameter is incorporated in each processor without waiting for all processors to do so.

5. They provide an improvement of the convergence thanks to the Gauss principle.

The major drawback of asynchronous algorithms is that they may diverge while their synchronous counterparts converge. Indeed, asynchronous iterations cannot be described mathematically by $x^{(k+1)} = T(x^{(k)})$.

Below, we give the mathematical model of asynchronous algorithms and we recall their convergence conditions.

5.2 Mathematical model and convergence results

5.2.1 The mathematical model of asynchronous algorithms

Suppose again that we have L processors and that an n-dimensional unknown vector is partitioned into L subvectors of dimension n_i, i.e., $n = \sum_{i=1}^{L} n_i$, so that each processor i can compute a vector of dimension n_i.

Consider the system of n equations

$$F(x) = 0, \tag{5.1}$$

and suppose that (5.1) has a unique solution x^*. Suppose that after some algebraic transformations the above system of equations is rewritten as

$$x = T(x). \tag{5.2}$$

Asynchronous executions of iterative algorithms associated to the above fixed point problem are described by the behavior of the following sequence (k denoting the k^{th} iteration)

$$\begin{cases} Given\ x^{(0)} = (x_1^{(0)}, ..., x_L^{(0)}) \\ for\ k = 0, 1, 2... \\ \quad for\ i = 1, ..., L \\ \quad\quad x_i^{(k+1)} = \begin{cases} T_i(x_1^{(\rho_1^i(k))}, ..., x_i^{(\rho_i^i(k))}, ..., x_L^{(\rho_L^i(k))}) & if\ i \in s(k) \\ x_i^{(k)} & if\ i \notin s(k), \end{cases} \end{cases} \tag{5.3}$$

where $S = \{s(k)\}_{k\in\mathbb{N}}$ is a sequence of nonempty subsets of $\{1, ..., L\}$. The subsets $s(k)$ represent the set of components updated at the iteration k; it is usually called the *steering* of the algorithm.

For $i \in \{1, ..., L\}$, $\rho^i = \{\rho^i_1(k), ..., \rho^i_L(k)\}_{k\in\mathbb{N}}$ is a sequence of integers such that:

$$\forall i, j \in \{1, ..., L\},\ \rho^i_j(k) \leq k,$$

where $\rho^i_j(k)$ represents the iteration number of the data coming from processor j and available on processor i at iteration k. In other words, the quantity $k - \rho^i_j(k)$ represents the delay of processor j according to processor i when the latter processor computes the i^{th} block at the k^{th} iteration. That delay could be due to the communication time or to the computation time of the j^{th} block.

The sequence (5.3) describes the behavior of iterative algorithms executed asynchronously on a parallel computer with L processors: at each iteration k, either the processor i computes $x_i^{(k+1)}$ by using the i^{th} component T_i (so the i^{th} component is updated) or it does not perform any computation and it keeps the value of $x_i^{(k)}$ computed at the previous iteration.

The two following conditions have to be ensured

$$card\{k \in \mathbb{N}, i \in s(k)\} = +\infty \tag{5.4}$$

and

$$\forall i, j \in \{1, ..., L\},\ \lim_{k\to+\infty} \rho^i_j(k) = +\infty. \tag{5.5}$$

Condition (5.4) corresponds, in a standard way, to the fact that none of the equations is definitively forgotten and so, none of the corresponding components is not refreshed after a certain rank of the iterations.

Condition (5.5) implies that old information is purged from the system. This results from the fact that any information is transmitted during a delay shorter than the one of a finite number r of refreshment of components. So

$$\rho^i_j(k) = k - r$$

and (5.5) is satisfied.

It should be noted that in the model (5.3), we do not need to have knowledge of the delays nor of the updated components; we only have to be sure that conditions (5.4) and (5.5) are satisfied.

Two asynchronous executions of the same algorithm do not give rise to the same iterations, but when the convergence of an asynchronous algorithm is proved, this means that it converges whatever the actual conditions of experimentation, provided that conditions (5.4) and (5.5) are satisfied. In what follows, we will refer to asynchronous algorithms associated to the successive approximations generated by a fixed point mapping T by $(T, Async)$.

5.2.2 Some derived basic algorithms

Asynchronous fixed point methods are not only a family of algorithms suitable for asynchronous computations on multiprocessors, but also a general framework allowing a general formulation of iteration methods associated to a fixed point mapping on a product space, including the most standard ones such as the successive approximation method (linear or nonlinear Jacobi method) and linear or nonlinear Gauss-Seidel method among many others.

If we take $\rho_j^i(k) = k$ for all processors i and j and all $k \in \mathbb{N}$, then (5.3) describes synchronous parallel algorithms.

If we take $\rho_j^i(k) = k$ for all processors i and j, all $k \in \mathbb{N}$ and

$$s(k) = \{1, ..., L\} \;\; for \; k \in \mathbb{N}$$

then (5.3) describes the successive approximation method applied to (5.2).

If we take $\rho_j^i(k) = k$ for all processors i and j, all $k \in \mathbb{N}$ and

$$s(k) = \{1 + k(mod)L\} \;\; for \; k \in \mathbb{N}$$

then (5.3) describes the synchronous Gauss-Seidel algorithm.

Suppose that each of the L processors deals with m equations so that

$$Lm = n$$

then to modelize the Gauss-Seidel method executed synchronously on the L processors, it is sufficient to take

$$\begin{cases} \forall k \in \mathbb{N}, \; s(k) = \bigcup_{l=1}^{L} \{(l-1)\,m + 1 + k(mod)m\} \\ \qquad \forall i \in \{1, 2, ..., n\}, \rho_j^i(k) = k. \end{cases}$$

This method corresponds to a situation associated to Gauss's principle, which asserts that any obtained partial result is immediately used anew. This algorithm corresponds to an ideal case in which not only each processor synchronously swaps with the others the m equations to which it is devoted, but also synchronously accesses produced results, independently of the fact that they are produced by itself or by another.

Suppose now that each processor runs a sequential Gauss-Seidel algorithm along the m equations to which it is devoted, and that the L processors run asynchronously. If we suppose that each processor has a local steering

$$s^k(ql) = (l-1)\,n + 1 + ql(mod)m$$

then we get the asynchronous Gauss-Seidel method.

It is worthwhile to note that in the general situation of asynchronous algorithms we do not know the global steering $s(k)$, nor the delays $\rho_j^i(k)$ of such algorithms and so we are unable to specify their exact formulations. But to study the behavior of an asynchronous algorithm it is not important to know the exact expression of the steerings and delays; we only have to be sure that this algorithm admits such a formulation of the form (5.3) and that conditions (5.4), (5.5) are satisfied.

5.2.3 Convergence results of asynchronous algorithms

In the case of a Banach product space (see the Appendix), the main convergence theorem in a contraction framework is El Tarazi's Theorem [50], [51], formulated as follows. Consider the following maximum weighted norm defined on a product Banach space $E = \prod_{i=1}^{L} E_i$ by

$$\begin{array}{c} for\ u = (u_1, ..., u_L) \in E = \prod_{i=1}^{L} E_i \\ \|u\|_{\gamma,\infty} = \max_{1 \le i \le L} \frac{|u_i|_i}{\gamma_i} \end{array} \tag{5.6}$$

where for $i \in \{1, ..., L\}$, $| \ . \ |_i$ is a norm defined on E_i and γ_i are positive real numbers.

THEOREM 5.1
Let T be a mapping from $D(T) \subset E$ in E and suppose that:
(a) $D(T) = \prod_{i=1}^{L} D_i(T)$
(b) $T(D(T)) \subset D(T)$
(c) $\exists u^* \in D(T)$, such that $u^* = T(u^*)$
(d) $\forall u \in D(T), ||T(u) - u^*||_{\gamma,\infty} \le \beta ||u - u^*||_{\gamma,\infty}$ with $0 < \beta < 1$,
then each asynchronous algorithm $(T, Async)$ associated to T converges to the fixed point u^* of T, whatever the starting point $u^0 \in D(T)$.

In fact, this theorem gives a generalization of a theorem due to Miellou [86]: Miellou's theorem is placed in a contraction matrix framework (with respect to a vectorial norm) which is included in the maximum weighted norm framework (with respect to a scalar norm). The weights γ_i are obtained by the Perron-Frobenius theorem in the case of monotone operators.

Bertsekas and Tsitsiklis established a more general convergence result based on nested sets [33]. This theorem is true in the general case of a Cartesian metric space. We give below its formulation in the case of the n-dimensional real space.

THEOREM 5.2
Let $E = \prod_{i=1}^{L} E_i \subset \prod_{i=1}^{L} \mathbb{R}^{n_i}$. Suppose that for each $i \in \{1, ..., L\}$, there exists a sequence of nested sets $E_i^{(k)}$ of subsets of E_i such that for all $k \ge 0$,

1. *$E_i^{(k+1)} \subset E_i^{(k)}$,*
2. *$T(E^{(k)}) \subset E^{(k+1)}$, where $E^{(k)} = \prod_{i=1}^{L} E_i^{(k)}$,*

then under assumptions (5.4) and (5.5) every limit point of the sequence $\{x^{(k)}\}_{k\in\mathbb{N}}$ generated by algorithm (5.3) and starting with $x^{(0)} \in E^{(0)}$ is a solution of the fixed point problem (5.2).

The last condition can be enlarged to the existence of attractors for the considered asynchronous iterations in the case of perturbed fixed point mapping, for example by round-off errors [87].

Convergence studies of asynchronous algorithms have been obtained by Chazan and Miranker [38] for linear systems, Miellou [86], Baudet [30], El Tarazi [50], [51] for contracting operators on the one hand and Miellou [86] and Bertsekas [32], for monotone iterations on the other hand. Uresin and Dubois [111] also have studied the convergence of asynchronous algorithms.

The sufficient conditions of Theorem 5.2 are shown to be necessary in the case where $n_i = 1$ for each i and T is a linear mapping ([38]).

Theorem 5.1 is in fact a particular situation which satisfies the general convergence result of Theorem 5.2. Indeed, if we consider the sets

$$E_i^{(k)} = \left\{ u_i \in \mathbb{R}^{n_i}, \ \ |u_i - u_i^*|_i \leq \beta^k \left| u_i^{(0)} - u_i^* \right|_i \right\},$$

with $x^{(0)} \in D(T) \subset E$, then one can easily verify that the subsets $E_i^{(k)}$ satisfy the conditions of Theorem 5.2.

In several practical situations, the exact solution of the fixed point equation $x_i^{(k+1)} = T_i(x_1^{(\rho_1^i(k))}, ..., x_i^{(k+1)}, ..., x_L^{(\rho_L^i(k))})$ in (5.3) cannot be obtained or its computation may be prohibitive, so that one has to approximate this solution by performing some iterations of a convergent process. In these situations, we are dealing with inner and outer iterations. The iterations are called *nonstationary* since T depends on the current outer iteration k ($T^{(k)}$). Theorem 5.2 is still valid by replacing T by $T^{(k)}$. The obtained model includes the so-called *two-stage algorithms.*

When the new computed values are sent to other processors as soon as they are computed and without waiting for the convergence, we are in the context of asynchronous iterations with flexible communications [65], [49]. The mathematical model for these algorithms can be obtained by introducing a second set of delays as follows:

$$\left\{ \begin{array}{l} Given\ x^{(0)} = (x_1^{(0)}, ..., x_L^{(0)}) \\ for\ k = 0, 1, 2... \\ \quad for\ i = 1, ..., L \\ \qquad x_i^{(k+1)} = \left\{ \begin{array}{ll} T_i\left((x_1^{(\rho_1^i(k))}, ..., x_L^{(\rho_L^i(k))}), (x_1^{(r_1^i(k))}, ..., x_L^{(r_L^i(k))}) \right) & if\ i \in s(k) \\ \qquad\qquad x_i^{(k)} & if\ i \notin s(k). \end{array} \right. \end{array} \right. \tag{5.7}$$

For a generalization of this model when the domain consists in multiple copies of E and each component of each copy is subject to different delays, see [64]. The following theorems ([65], [64]) which can be considered as a generalization of Theorem 5.1 give sufficient convergent conditions of asynchronous algorithms with flexible communications.

THEOREM 5.3
Assume that there exists $x^* \in E$ such that $T(x^*, x^*) = x^*$ and assume that there exist $\gamma \in [0, 1[$ and a weighted maximum norm such that for all $x, y \in E$,

$$||T(x,y) - x^*||_{\gamma,\infty} \leq \gamma \max(||x - x^*||_{\gamma,\infty}, ||y - x^*||_{\gamma,\infty}),$$

then the asynchronous iterations described by (5.7) converges to x^*.

Asynchronous algorithms which satisfy conditions (5.4) and (5.5) are called *totally asynchronous algorithms* in opposition to partial asynchronous ones. The difference between the two kinds of algorithms mainly lies in the assumptions made on the delays between the processors. Condition (5.5) is replaced by the following conditions.

There exists a positive integer B such that for every iteration k, we have

$$\forall i, j \in \{1, ..., L\},\ k - B + 1 \leq \rho_j^i(k) \leq k \tag{5.8}$$

$$\forall i \in \{1, ..., L\},\ \rho_i^i(k) = k \tag{5.9}$$

Note that condition (5.8) means that old information is purged from the network after at most B iterations. This condition is satisfied in practice. Condition (5.9) means that the own computed components of a processor are never outdated. This last condition is also generally satisfied.

5.3 Convergence situations

5.3.1 The linear framework

Consider again the linear system

$$Ax = b, \tag{5.10}$$

where A is a $n \times n$ square nonsingular matrix and let

$$A = M - N \tag{5.11}$$

be a splitting of A, i.e., M is a nonsingular matrix. Consider the iterative algorithm associated to the splitting (5.11) and defined by

$$\begin{cases} x^{(0)}\ given \\ x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b,\ for\ all\ k \in \mathbb{N} \end{cases} \tag{5.12}$$

Let $T = M^{-1}N$ and $|T|$ denotes the matrix whose entries are the absolute values of the entries of T. Then we have the following result due to Chazan and Miranker [38].

THEOREM 5.4

1. *If $\rho\left(|T|\right) < 1$ ($|T| = \left(|T_{i,j}|\right)_{i,j}$) then the asynchronous iterations (5.12) converge to the solution of (5.10).*
2. *If $\rho\left(|T|\right) \geq 1$ there exists a set of delays and strategies and an initial guess $x^{(0)}$ such that the corresponding asynchronous iteration does not converge to the solution of (5.10).*

The proof of this theorem derives from the Perron-Frobenius theorem which states that $\rho\left(|T|\right) < 1$ if and only if T is a contraction with respect to a weighted maximum norm of the form (5.6).

In the next part of this section we are interested in the case of M-matrices (see the Appendix). Let us suppose that $A = (A_{i,j})$ is an M-matrix and consider the by point Jacobi splitting of A,

$$A = D - B,$$

where $D = diag(..., A_{i,i}, ...)$. Remark that as A is an M-matrix, B is a non-negative matrix. Consider the by point Jacobi iterations associated to the fixed point mapping G defined by

$$\forall x \in \mathbb{R}^n,\ y = G(x) \Leftrightarrow y = D^{-1}Bx + D^{-1}b,$$

then we have the following result which is proved for example in [26].

PROPOSITION 5.1

G is contractive with respect to a weighted maximum norm of the form (5.6), where the vector γ is obtained from the Perron-Frobenius theory for nonnegative matrices and the constant of contraction of G is either $\rho(D^{-1}B)$ if $D^{-1}B$ is irreducible or $\rho(D^{-1}B) + \varepsilon$ (for any $\varepsilon > 0$) if $D^{-1}B$ is reducible.

The next result, due to Bahi and Miellou [26], can be considered as an extension of the classical Stein-Rosenberg theorem [108] in the case of asynchronous algorithms. It allows us to build asynchronous convergent algorithms and to compare their speed of convergence.

Let $A = M - N$ be a regular per block splitting of A and T be the fixed point mapping corresponding to this splitting, i.e.,

$$\forall x \in \mathbb{R}^n,\ y = T(x) \Leftrightarrow y = M^{-1}Nx + M^{-1}b.$$

Consider also T_ω the relaxed fixed point mapping defined by

$$\forall x \in \mathbb{R}^n,\ y = T_\omega(x) \Leftrightarrow y = (1 - \omega)x + T(x),$$

then

PROPOSITION 5.2
If $\omega \in \left]0, \frac{2}{1+\rho(M^{-1}N)}\right[$ then any asynchronous algorithm associated to T_ω converges to the solution of (5.10). Moreover,

$$\rho(M^{-1}N) < \rho(D^{-1}B) < 1.$$

It should be noticed that, as stated in [26], the above proposition is also true in the case where A is an H-matrix with positive diagonal elements.

5.3.2 The nonlinear framework

Suppose that in the system of equations (5.1), F is a nonlinear mapping and that there exists a unique solution x^* on $D(F) \subset \mathbb{R}^n$. Suppose also that we have an iterative algorithm whose convergence is described by a fixed point mapping T which satisfies $x^* = T(x^*)$, then in [50] we have the following result on the local convergence of asynchronous iterations associated to T. This theorem can be considered as a generalization of the Ostrowski theorem [95] on successive iterations.

THEOREM 5.5
Assume that T is Fréchet differentiable at x^* and that x^* lies on the interior of $D(F)$. If $\rho(|T'(x)|) < 1$ then there exists a neighborhood V_{x^*} of x^* such that any asynchronous iteration associated to T and started with $x^{(0)} \in V_{x^*}$ converges to x^*.

An important iterative algorithm to solve (5.1) is the Newton algorithm for which

$$T(x) = x - F'(x)^{-1}F(x).$$

As $T'(x^*) = 0$, the above theorem can be applied.

In practice, the computation of $F'(x)^{-1}$ is expensive so one may use the quasi Newton methods by replacing $F'(x)^{-1}$ by a more simple computation. The above theorem is then applied by considering the spectral radius of the new $|T'(x)|$.

5.4 Parallel asynchronous multisplitting algorithms

In this section we come back to parallel multisplitting algorithms and we analyze their convergence when they are executed asynchronously on a grid environment. Multisplitting algorithms are well suited for parallel computing because each splitting gives rise to a subproblem which can be solved by a processor. Parallel asynchronous multisplitting algorithms may have several advantages. Indeed, in general, when the ratio communication time/computation time is not negligible, asynchronous execution may reduce the total time of computation by cancelling the idle times due to synchronizations between the iterations.

Another important point of multisplitting algorithms is that a processor may use a direct solver which is adapted to its problem independently from the other processors, which induces an interesting coupling of different solvers to treat a large problem on a grid environment.

In the next part of this section we follow the paper of Bahi et al. [27] and we give a unified mathematical framework of asynchronous multisplitting algorithms and then we consider the linear and nonlinear contexts.

5.4.1 A general framework of asynchronous multisplitting methods

We consider problems of the form (5.1)

$$F(x) = 0,\ x \in D \subset \mathbb{R}^n$$

where F is a nonlinear operator defined on a closed set D where

$$D = \prod_{i=1}^{n} d_i \text{ and } d_i \subset \mathbb{R} \text{ are closed and convex.} \tag{5.13}$$

Suppose that the solution of (5.1) is x^*. Suppose also the existence of L mappings $T^{(l)}$ on D such that

$$T^{(l)}(D) \subset D \tag{5.14}$$

and

$$T^{(l)}(x^*) = x^*,\ \forall l \in \{1, ..., L\} \tag{5.15}$$

Assume that for $l \in \{1, ..., L\}$, $T^{(l)}$ is contractive with respect to x^* and to a norm $|\ .\ |_{\infty,\gamma}$,

$$\begin{cases} |x|_{\infty,\gamma} = \max\limits_{1 \le i \le n} \dfrac{|x_i|}{\gamma_i} \\ \gamma_i > 0. \end{cases} \tag{5.16}$$

Here, $|x_i|$ denotes the absolute value of x_i in $\mathbb{R}$, i.e.,

$$\left| T^{(l)}(x) - x^* \right|_{\infty,\gamma} \le \nu_l \left| x - x^* \right|_{\infty,\gamma} \tag{5.17}$$

DEFINITION 5.1 *A formal multisplitting associated to (5.1) is a collection of fixed point problems*

$$x - T^{(l)}(x) = 0,\ l \in \{1, ..., L\}$$

where each $T^{(l)}$ satisfies the conditions (5.14), (5.15) and (5.17). Let us fix the following notations,

$$x^l \textit{ and } x^k,\ l,k \in \{1,...,L\}$$

are vectors of $\mathbb{R}^n$ the components of which are

$$x_i^l \textit{ and } x_j^k,\ i,j \in \{1,...,n\}.$$

Define the extended fixed point mapping

$$\begin{cases} \mathcal{T} : (\mathbb{R}^n)^L & \longrightarrow \quad (\mathbb{R}^n)^L \\ X = (x^1,...,x^L) & \longmapsto Y = (y^1,...,y^L) \end{cases}$$

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

$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum\limits_{k=1}^{L} E_{lk}(X)x^k \end{cases} \tag{5.18}$$

where $E_{lk}(X)$ are weighting matrices satisfying

$$\begin{cases} E_{lk}(X) \textit{ are diagonal matrices} \\ E_{lk}(X) \geq 0 \\ \sum\limits_{k=1}^{L} E_{lk}(X) = I_n \ \ (\textit{identity matrix in } \mathbb{R}^n),\ \forall l \in \{1,...,L\} \end{cases} \tag{5.19}$$

Since $T^{(l)}(D) \subset D$ we have

$$\mathcal{T}(U) \subset U \tag{5.20}$$

where $U = \prod\limits_{i=1}^{L} D$.

Then the successive approximations associated to the extended fixed point mapping $\mathcal{T}$ describe the behavior of any multisplitting algorithms associated to (5.1).

As mentioned in [27] and in the previous chapter, the dependence of the weighting matrices $E_{lk}(X)$ on l, k and the current element X allows us:

- to take $E_{lk}(X) = E_k$ in order to obtain O'Leary and White multisplitting algorithms, as seen in Chapter 4.
- to define $E_{lk}(X) = E_{lk}$ depending on the index l in order to give a presentation of either the Schwarz alternating method or the general Schwarz multisplitting methods, as seen in Chapter 4.
- to take $E_{lk}(X)$ depending on both the index l and on the element X of $(\mathbb{R}^n)^L$, the value of which must be the current iterate X^p in order to describe two-stage multisplitting methods.

Under the assumptions of (5.14), (5.17), (5.19) we have the following result on $\mathcal{T}$:

PROPOSITION 5.3
Denote $X^* = (x^*, ..., x^*)$ where x^* is the solution of (5.1), then $\mathcal{T}$ is contractive with respect to X^* and to $| \, . \, |_{\infty,\gamma}$ which is defined by

$$|X|_{\infty,\gamma} = \max_{1\le k\le L} \max_{1\le i\le n} \frac{\left|\left(x^k\right)_i\right|}{\gamma_i} \tag{5.21}$$

Its constant of contraction is

$$\nu = \max_{1\le l\le L} \nu_l \tag{5.22}$$

and X^* is the fixed point of $\mathcal{T}$.

PROOF Take any $Y = \mathcal{T}(X)$, by (5.18) we have

$$\left|y^l - x^*\right|_{\infty,\gamma} = \left|T^{(l)}\left(\sum_{k=1}^{L} E_{lk}(X)x^k\right) - x^*\right|_{\infty,\gamma}$$

we have

$$\frac{\left|\left(\sum\limits_{k=1}^{L} E_{lk}(X)\left(x^k - x^*\right)\right)_i\right|}{\gamma_i} = \frac{\left|\sum\limits_{k=1}^{L}\sum\limits_{j=1}^{n} \left(E_{lk}(X)\right)_{i,j}\left(x^k - x^*\right)_j\right|}{\gamma_i}$$

Since the weighting matrices $E_{lk}(X)$ are diagonals, we have

$$\left|\sum_{j=1}^{n} \left(E_{lk}(X)\right)_{i,j}\left(x^k - x^*\right)_j\right| = \left|\left(E_{lk}(X)\right)_{i,i}\left(x^k - x^*\right)_i\right|$$

condition (5.19) gives

$$\left|\sum_{k=1}^{L} \left(E_{lk}(X)\right)_{i,i}\left(x^k - x^*\right)_i\right| \le \underbrace{\sum_{k=1}^{L} \left(E_{lk}\right)_{i,i}(X)}_{1} \max_{1\le k\le L} \left|\left(x^k - x^*\right)_i\right|$$

so

$$\max_{1\le i\le n} \frac{\left|\left(T^{(l)}\left(\sum\limits_{k=1}^{L} E_{lk}(X)x^k\right) - x^*\right)_i\right|}{\gamma_i} \le \nu_l \max_{1\le i\le n} \max_{1\le k\le L} \frac{\left|\left(x^k - x^*\right)_i\right|}{\gamma_i}$$

then

$$\max_{1\le l\le L}\max_{1\le i\le n}\frac{|(y^l-x^*)_i|}{\gamma_i}\le\max_{1\le l\le L}\left(\nu_l\max_{1\le i\le n}\frac{\left|\left(x^l-x^*\right)_i\right|}{\gamma_i}\right)$$

by (5.21) and (5.22) we have

$$|Y-X^*|_{\infty,\gamma}\le\nu\,|X-X^*|_{\infty,\gamma}$$

since $\sum_{k=1}^{L}E_{lk}(X)=I_n$ and $x^*=T^{(l)}(x^*)$ we have $\mathcal{T}(X^*)=X^*$. ▯

The above result gives the important following general convergence result of asynchronous multisplitting algorithms described by the fixed point mapping $\mathcal{T}$ for solving (5.1).

COROLLARY 5.1
Under the assumptions of Proposition 5.3, any asynchronous algorithm $(\mathcal{T}, Async)$, corresponding to $\mathcal{T}$ and starting with $X^0\in U$, converges to the solution of (5.1).

PROOF Condition (5.20) implies that $\mathcal{T}$ has a unique fixed point; Theorem 5.1 and Proposition 5.3 end the proof. ▯

5.4.2 Asynchronous multisplitting algorithms for linear problems

In Chapter 4, we have shown how to build convergent synchronous multisplitting algorithms by splitting the nonsingular square matrix A of the linear system. We will now give a convergence result on asynchronous multisplitting algorithms for the solution of linear systems.

Consider again, as in Section 5.3.1, the linear system (5.10)

$$Ax=b,$$

where A is a $n\times n$ square nonsingular matrix and consider L splittings of A which are supposed to be regular

$$A=M_l-N_l,\ l=1,...,L$$

Then we can build a multisplitting as in definition (5.1) by setting

$$T^{(l)}(x)=M_l^{-1}N_lx+M_l^{-1}b.$$

Thus the successive approximations associated to the extended fixed point mapping $\mathcal{T}$ defined in (5.18) describe the behavior of parallel multisplitting algorithms for the solution of (5.10).

Process (5.3), where T is replaced by $\mathcal{T}$, describes the behavior of asynchronous execution of parallel multisplitting algorithms. A simple application of Proposition 5.2 shows that, for example, if A is an M-matrix then T_l are contractive mappings and as a consequence of Proposition 5.3 and Corollary 5.1 we obtain the following convergence result.

PROPOSITION 5.4

If the matrix A of the linear system (5.10) is an M-matrix, then any asynchronous multisplitting algorithm, associated with regular splittings of A, converges to the solution of the linear system (5.10).

By a suitable choice of the weighting matrices E_{lk} we can, as seen for the parallel synchronous case in Chapter 4, define asynchronous versions of the O'Leary and White and Schwarz multisubdomain algorithms.

5.4.3 Asynchronous multisplitting algorithms for nonlinear problems

In this section we introduce a tool which allows us to build splittings for either linear or nonlinear fixed point equations.

For a more general tool to build splittings of a nonlinear problem, the interested reader should consult [27]. We consider problems in the form (5.1),

$$F(x) = 0$$

which can be rewritten in the fixed point equation form (5.2),

$$x = T(x),\ x \in D \subset \mathbb{R}^n,$$

where D satisfies (5.13) and T satisfies the contraction assumption

$$\begin{cases} |T(x) - T(y)|_{\infty,\gamma} \leq \nu\, |x - y|_{\infty,\gamma} \\ 0 < \nu < 1 \end{cases}$$

and

$$T(D) \subset D \tag{5.23}$$

Let I_l, $l \in \{1, ..., L\}$, be subsets of $\{1, ..., n\}$ and I_l^C their complementaries

$$I_l \cup I_l^C = \{1, ..., n\}\,, \forall l \in \{1, ..., L\} \tag{5.24}$$

and define the vectors $\sigma_i^l\,(u, v)$ by

$$\begin{cases} \sigma_i^l(u, v) = (w_1, ..., w_n) \text{ such that} \\ w_j = u_j\ if\ \ (i, j \in I_l)\ \text{ or }\ \left(i, j \in I_l^C\right) \\ w_j = v_j \text{ otherwise} \end{cases} \tag{5.25}$$

DEFINITION 5.2 *The block $\left(I_l, I_l^C\right)$ splittings are defined by the following mappings F_l*

$$\forall x \in D, \forall i \in \{1,...,n\}, T_i^{(l)}(x) = T_i\left(\sigma_i^l\left(T^{(l)}(x), x\right)\right) \tag{5.26}$$

In such a case we usually take

$$(E_{lk})_{i,j} = 0 \; or \; (E_k)_{i,j} = 0 \; for \; j \in I_k^C \tag{5.27}$$

It should be noticed that if we except particular problems which admit a natural block decomposition structure suitable for block iterative algorithms, the previous condition (5.27) is very important, especially for overlapping block decomposition techniques, because in the evaluation of $T^{(l)}$, for any k we never have to use any component the index of which lies in I_k^C, so in the block $\left(I_l, I_l^C\right)$ splitting the solution of a diagonal block subproblem associated to any I_l^C never has to be computed.

PROPOSITION 5.5
For $l \in \{1,...,L\}$, $T^{(l)}$ is $|\;.\;|_{\infty,\gamma}$ contractive, its constant is less than or equal to the constant of T and the fixed point of $T^{(l)}$ is x^*.

PROOF

$$\frac{\left|T_i^{(l)}(x) - T_i^{(l)}(y)\right|}{\gamma_i} = \frac{\left|T_i\left(\sigma_i^l\left(T^{(l)}(x), x\right)\right) - T_i\left(\sigma_i^l\left(T^{(l)}(y), y\right)\right)\right|}{\gamma_i}$$

$$\leq \nu \left|\sigma_i^l\left(T^{(l)}(x), x\right) - \sigma_i^l\left(T^{(l)}(y), y\right)\right|_{\infty,\gamma}$$

$$\leq \nu \max\left(\max_{1\leq j\leq n} \frac{\left|T_j^{(l)}(x) - T_j^{(l)}(y)\right|}{\gamma_j}, \max_{1\leq j\leq n} \frac{|x_j - y_j|}{\gamma_j}\right)$$

so either

$$\frac{\left|T_i^{(l)}(x) - T_i^{(l)}(y)\right|}{\gamma_i} \leq \nu \max_{1\leq j\leq n} \frac{\left|T_j^{(l)}(x) - T_j^{(l)}(y)\right|}{\gamma_j}$$

which implies that $\left|T_i^{(l)}(x) - T_i^{(l)}(y)\right| = 0$
or

$$\frac{\left|T_i^{(l)}(x) - T_i^{(l)}(y)\right|}{\gamma_i} \leq \nu \max_{1\leq j\leq n} \frac{|x_j - y_j|}{\gamma_j}$$

so

$$\left|T^{(l)}(x) - T^{(l)}(y)\right|_{\infty,\gamma} \leq \nu\, |x - y|_{\infty,\gamma}$$

which implies that $T^{(l)}$ is contractive and that its constant is less than or equal to ν.

Moreover by (5.23) F and T have an unique fixed point; let

$$T_i^{(l)}(x) = x_i$$

so equivalently

$$x_i = T_i\left(\sigma_i^l\,(x, x)\right)$$

so

$$x = T\,(x)$$

hence $T^{(l)}$ and T have the same fixed point x^*. □

5.4.3.1 Extended fixed point mapping associated with $\left(I_l, I_l^C\right)$ multisplitting

Take the diagonal positive matrices $E_{lk}(X)$ depending only on k

$$E_{lk}(X) = E_k$$

and satisfying

$$\begin{cases} \sum\limits_{k=1}^{L} E_k = I_n \\ (E_k)_{i,i} = 0, \ \forall i \notin I_k \end{cases} \tag{5.28}$$

The asynchronous iterations corresponding to $\left(I_l, I_l^C\right)$ multisplitting are defined by the fixed point mapping

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

$$\begin{cases} y^l = T^{(l)}(z) \\ z = \sum\limits_{k=1}^{L} E_k x^k \end{cases} \tag{5.29}$$

where for $l \in \{1, ..., L\}$, $T^{(l)}$ is defined by (5.26). We remark that this multisplitting algorithm is analogous to O'Leary and White multisplitting algorithms for nonlinear problems.

As a consequence of Propositions 5.3 and 5.5 we have

COROLLARY 5.2

Any asynchronous algorithm ($\mathcal{T}^{OW}$, Async) corresponding to $\mathcal{T}^{OW}$ and starting with $X^0 \in U$ converges to the solution of (5.1).

5.4.3.2 The discrete analogue of Schwarz alternating method and its multisubdomain generalizations

Asynchronous Schwarz alternating methods and their multisubdomain generalizations are obtained by choosing the weighted matrices exactly as in Chapter 4. In the following, we point out, once again, these choices.

5.4.3.3 Discrete analogue of the Schwarz alternating method

Suppose $I_1 \bigcap I_2 \neq \emptyset$, so we have an overlap between the 1st and the 2nd subdomains. Consider the matrices E_{lk} such that

$$(E_{11})_{i,i} = \begin{cases} 1 \ \forall i \in I_1 \\ 0 \ \forall i \notin I_1 \end{cases}, \quad (E_{12})_{i,i} = \begin{cases} 0 \ \forall i \in I_1 \\ 1 \ \forall i \notin I_1 \end{cases} \tag{5.30}$$
$$(E_{21})_{i,i} = \begin{cases} 1 \ \forall i \notin I_2 \\ 0 \ \forall i \in I_2 \end{cases}, \quad (E_{22})_{i,i} = \begin{cases} 0 \ \forall i \notin I_2 \\ 1 \ \forall i \in I_2 \end{cases}$$

Define the fixed point mapping

$$\mathcal{T}^S(x^1, x^2) = (y^1, y^2) \ such\ that\ for\ l = 1,2$$
$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum_{k=1}^{2} E_{lk} x^k \end{cases} \tag{5.31}$$

where for $l \in \{1,2\}$, $T^{(l)}$ is defined by (5.26). 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 $\mathcal{T}^S$. For the use of such methods as preconditioners of Krylov spaces methods, we refer to [72], [106].

5.4.3.4 Discrete analogue of the multisubdomain Schwarz method

We introduce the weighting matrices E_k satisfying (5.28) and the matrices E_{lk} such that for $l \in \{1, ..., L\}$

$$(E_{ll})_{i,i} = \begin{cases} 1 \ if\ i \in I_l \\ 0 \ if\ i \notin I_l \end{cases}$$
$$(E_{lk})_{i,i} = \begin{cases} 0 \ if\ i \in I_l \\ (E_k)_{i,i} \ \ if\ i \notin I_l \end{cases} \tag{5.32}$$

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

$$\mathcal{T}^{MS}(x^1, ..., x^L) = (y^1, ..., y^L) \ such\ that$$
$$\begin{cases} y^l = T^{(l)}(z^l) \\ z^l = \sum_{k=1}^{L} E_{lk} x^k \end{cases} \tag{5.33}$$

where E_{lk} are defined by (5.32) and $T^{(l)}$ are defined by (5.26).

$\mathcal{T}$ being $\mathcal{T}^{OW}$ or $\mathcal{T}^S$ or $\mathcal{T}^{MS}$ we have the following Corollary.

COROLLARY 5.3
Any asynchronous algorithm $(\mathcal{T}, Async)$, corresponding to $\mathcal{T}$ and starting with $X^0 \in U$, converges to the solution of (5.1).

5.5 Coupling Newton and multisplitting algorithms

The standard algorithm for solving the system of nonlinear equations (5.1) is the Newton algorithm; an effective way to use the Newton algorithm in a parallel environment is to couple it with multisplitting algorithms.

There are two ways to realize this coupling. The first one consists in splitting the linear problems involved in each iteration of the Newton algorithm and the second one consists in splitting the nonlinear problem (5.1) itself into subproblems and solving each subproblem using the Newton algorithm. Below, we describe the algorithmic formulation of these two kinds of mixed Newton multisplitting algorithms.

5.5.1 Newton-multisplitting algorithms: multisplitting algorithms as inner algorithms in the Newton method

Recall that the Newton algorithm for solving the nonlinear system of equation (5.1), $F(x) = 0$ is described by the iterations

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

As in Chapter 4, we will suppose that (5.1) has a solution x^*, that F is Fréchet differentiable on a neighborhood of x^* and that F' is nonsingular and Lipschitz continuous on a neighborhood of x^*. We have seen in Chapter 4 that the Newton method involves the solution of a linear system

$$F'(x^{(k)})y = F(x^{(k)}) \tag{5.34}$$

and that this solution allows the computation of the next Newton iterates $x^{(k+1)}$ by setting $y^{(k)} = y$ in the following equation:

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

The solution of (5.34) by splitting $F'(x^{(k)})$ gives rise to the multisplitting methods to solve this kind of problem. We call the global algorithm to solve (5.1) the *Newton-multisplitting* algorithm.

So, suppose we have L processors and that, as explained in Chapter 4, we have L splittings of $F'(x^{(k)})$ at each iteration k, so that we have

$$F'(x^{(k)}) = M_l(x^{(k)}) - N_l(x^{(k)}),\ l = 1, ..., L. \tag{5.35}$$

For simplicity sake, suppose that the weighting matrices only depend on one index and that the solution of system (5.34) is approximated by performing q iterations of the multisplitting method.

The parallel Newton-multisplitting method can be defined as follows

$$x^{(k+1)} = G(x^{(k)}), \tag{5.36}$$

where

$$G(x) = x - A(x)F(x), \tag{5.37}$$

and

$$A(x) = \sum_{l=1}^{L} E_l(x) \sum_{j=0}^{q-1} (M_l(x)^{-1} N_l(x))^j M_l(x)^{-1}.$$

If we take $y^{(0)} = 0$, then

$$A(x) = \sum_{l=1}^{L} E_l(x)(I - (M_l(x)^{-1} N_l(x))^q (F'(x))^{-1}. \tag{5.38}$$

THEOREM 5.6

If the splittings (5.35) are weak regular convergent, then there exists a neighborhood V_{x^*} of the solution x^* such that any asynchronous Newton-Multisplitting algorithm associated to (5.37) and (5.38), and starting from $x^{(0)} \in V_{x^*}$ converges to x^*.

PROOF We apply Theorem 5.5. We have

$$G'(x^*) = I - A(x^*)F'(x^*). \tag{5.39}$$

From (5.38) we have

$$G'(x^*) = I - \sum_{l=1}^{L} E_l(x^*)(I - (M_l(x^*)^{-1} N_l(x^*))^q. \tag{5.40}$$

The properties of the weighting matrices imply that

$$|G'(x^*)| = G'(x^*) = \sum_{l=1}^{L} E_l(x^*)(M_l(x^*)^{-1} N_l(x^*))^q. \tag{5.41}$$

As the splittings (5.35) are convergent, we deduce by the application of Proposition 3.2 of [27] that

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

The result follows from Theorem 5.5. ∎

There exist particular situations which satisfy the assumptions of the above convergence result. For example, if $F'(x)$ is monotone (i.e., $F'(x)^{-1} \ge 0$) then every weak regular splitting of $F'(x)$ is convergent [31].

5.5.2 Nonlinear multisplitting-Newton algorithms

Another way to mix the Newton method and the multisplitting approach is to use the result of Bahi et al. [26], [27] on nonlinear multisplitting. Indeed, the Newton method applied to the nonlinear problem (5.1) is described by the iterations associated to the contractive fixed point mapping

$$x = T(x),\ x \in \mathbb{R}^n,$$

where

$$T(x) = x - F'(x)^{-1}F(x).$$

Consider now L subsets I_l of $\{1, ..., n\}$ and weights E_{lk}, $l, k \in \{1, ..., L\}$, then Definition 5.2 of Section 5.4.3 allows us to generate L splittings of (5.2),

$$x = T^{(l)}(x),\ x \in \mathbb{R}^n,$$

with

$$x_i = T_i\left(\sigma_i^l\left(T^{(l)}(x), x\right)\right).$$

The application of Proposition 5.5 implies that $T^{(l)}$ are contractive mappings, so they define a nonlinear formal multisplitting: $x - T^{(l)}(x)$, $l \in \{1, ..., L\}$ as in Definition 5.1. We consider asynchronous algorithms associated to those splittings and weights E_{lk}. We call those algorithms *nonlinear multisplitting-Newton* algorithms.

Practically, the iterations, generated by each fixed point mapping $T^{(l)}$ defined just above, correspond to the iterations generated by the Newton algorithm and applied to a subproblem of (5.1). These subproblems are defined by the (I_l, I_l^C) splittings; they correspond to the computation of $card(I_l)$ components of x. We then have the convergence result which is a consequence of Proposition 5.3 and Corollary 5.1.

PROPOSITION 5.6
Suppose that the Newton algorithm with the initial guess $x^{(0)} \in V_{x^*}$ converges to x^*, a solution of (5.1) in $V_{x^*} \subset D(F)$, then the nonlinear multisplitting-Newton algorithm started with $x^{(0)}$ converges to x^*.

5.6 Implementation

The implementation of an asynchronous iterative algorithm may seem easier to achieve since there is no more synchronization. Nevertheless, as described in the previous section, the convergence detection is different and is not the most trivial point to implement in a distributed environment. Indeed, according to the dedicated architecture, a centralized mechanism can either be used (for a parallel architecture or a cluster with a high speed network with a quite limited number of processors) or is completely inconceivable (with a distributed cluster or a grid with a large number of processors). In Section 3.2.1, the classification of parallel iterative algorithms points out another crucial point allowing us to distinguish a synchronous parallel iterative algorithm from an asynchronous one. It deals with the communications management. With a synchronous algorithm, all messages sent are received and used. With an asynchronous algorithm, according to the implementation of communications management, it may not be the case. Indeed, on the sending side, some messages may not be actually sent if newer local data are available before their emission. Likewise, on the receiving side, some messages may not be taken into account if newer messages arrived before their use. Furthermore, it should be remembered that asynchronous iterative algorithms support message loss.

In Chapter 4, we have detailed the parallelization of some well-known algorithms. Only some of them can be executed using asynchronous iterations. For example, it is not possible to execute a parallel Conjugate Gradient, or a GMRES with asynchronous iterations. Roughly speaking, only algorithms based on the Jacobi method and the multisplitting method can be executed in an asynchronous mode.

Before explaining those algorithms, it is essential to review the different ways to manage the asynchronism in programming and execution environments. With AIAC algorithms, the iterations are asynchronous and so are the communications. Consequently, communications must be dissociated from the computations. With several traditional parallel environments based on the message passing paradigm (like PVM [66], MPI [71]), it is possible to use buffered sendings and nonblocking receptions. Nevertheless all emitted messages must be received using a receive operation. One of the particularities of AIAC algorithms is that when there are several versions of the same message (corresponding to different iterations), the program should only take the last version in order to converge faster. To clarify this, let us take a simple example. Consider that two processors are executing an AIAC algorithm and that processor 1 performs its iterations two times faster than processor 2. Consider also, that at each iteration processor 2 receives on average two messages from processor 1 and that processor 1 only receives a message from processor 2 approximately one iteration out of two. If the processors only test an iteration once, if a message arrives, then processor 2 would have a lot of delay in the reception of messages since at each iteration k it would approximately have k messages in delay. Of course, it is not possible to know a priori the number of messages that a processor will receive at each iteration and this number varies from one iteration to another. So, in a traditional message passing based environment, a naive solution consists in receiving all messages at each iteration and only using the last one. Then, the problem comes from the convergence detection that must be very efficient, and for that, as soon as a message for the convergence arrives it must be detected. That is why it is essential to dissociate the communications from the computations. For that, the only solution from our point of view lies in using a multithreaded environment which allows us to execute the computations in one thread and the management of the communications in other threads.

In order to keep the same formalism, we do not focus on the implementation of AIAC algorithms with shared memory architectures. With such environments, as soon as a mechanism to simulate communications between AIAC algorithms has been implemented, the following algorithms are quite easy to adapt.

5.6.1 Some solutions to manage the communications using threads

According to the flexibility of the communications in an AIAC algorithm, it is possible to distinguish different levels of communications management. The simplest solution, from the programmer point of view, consists in using an environment suited to the design of AIAC algorithms. Currently two programming environments fulfill those requirements, namely, JACE [23, 22, 24, 19] and CRAC [40]. Both environments have been developed in order to provide a communication library that allows us to design synchronous and asynchronous iterative algorithms. They use two queues that are executed into two threads: one for the message sendings and another one for the receptions. According to the execution mode (synchronous or asynchronous), the operating of those queues is different. In the synchronous mode, those queues are managed traditionally, i.e., when a computation task needs to send a message, the message is put in the sending queue that actually sends it, the reception queue receives it on the destination processor and the computation task on the other machine can use the message. In the asynchronous mode, the sending queue first checks whether a similar version of the message is not already in the queue (based on its tag, sender and receiver). In this case, the previous one is replaced by the newest one. The reception queue acts similarly when receiving a message. It checks the reception queue and replaces an old message by a recent one whenever possible. So, when a computation task receives a message, it is ensured to have the latest version available. Of course, in JACE and CRAC, the programmer does not need to interact with the threads which transparently manage the communications. Those two environments are detailed in Sections 6.2.1 and 6.2.2.

With a multithreaded version of MPI [7], or Corba [98] or PM2 [89] which are implicitly multithreaded, it is possible to implement AIAC algorithms. However, this requires a stronger endeavor from the programmer point of view since the management of threads is explicit. Using an environment with explicit management of threads, a programmer may assign one or more threads in charge of sending some messages and as many in charge of receiving them. Although the endeavor is stronger, it allows us to manage more precisely the communications. For example, it is possible to implement flexible models defined in Section 3.4.3. The sending of a message is as flexible as using any message passing interface since a user can send a message anywhere in a program. However, using a thread that can directly handle a message as its reception occurs is a possible source of convergence speed-up. As network resources in a distant environment often are a critical point, a possible strategy consists in assigning a thread to each destination neighbor and in waiting for the previous message to have arrived at its destination before sending another one. For that, the use of a mutex combined with an acknowledgment message allows the programmer to control the sending of each message. The principle is the following: when a processor wants to send a message to a given neighbor, the thread that is dedicated to this neighbor is locked (unless it was already locked and in that case, the message is not sent). Then the message is sent, the thread on the emitter processor is blocked until the neighbor confirms that it has received the message. When the emitting thread has received the acknowledgement of reception, it unlocks the mutex. So, it is then ready to send another message. If another message was supposed to be sent, then the mutex would be locked so the sending would not be possible. As a consequence, the network would not be overloaded with a useless message, since the previous one would not have been handled yet. One of the drawbacks of this explicit management of threads is that when the number of neighbors per processor is not known in advance or is dynamic, it is difficult to define a number of threads a priori. Moreover, that difficulty comes for both the sending and the reception. Furthermore, the explicit management of threads requires much more attention than traditional programming because they may lead to deadlock situations if the programmer is not very attentive. When the number of threads running simultaneously becomes too important, the scheduling may be less fair, which is not acceptable. In fact, the fairness is an essential requirement of the threads management since the convergence conditions of AIAC algorithms involve that each processor should be able to regularly update its components.

In the following we present some asynchronous iterative algorithms in which we consider that messages arrive in their emitted order. If this is not the case, a simple mechanism should be added which consists in adding the iteration number at which the sent data have been produced on the sender. Then, on the receiver, the iteration number included in the message is compared to the one of the last message taken into account from that source. Finally, if the number in the message is smaller than the current one, the message is suppressed without being taken into account. Otherwise, the message is used and the current iteration number related to that source is updated. Implementing that mechanism allows us to ensure a faster convergence.

5.6.2 Asynchronous Jacobi algorithm

The asynchronous version of the synchronous Jacobi algorithm presents several similarities with it. In fact only two parts are different, the management of the communications and the convergence detection. In Algorithm 5.1 we give a possible implementation of the asynchronous Jacobi. With such a formalism which hides the mechanism to manage the asynchronism, it is quite easy to write the asynchronous version starting with the synchronous one. In this algorithm, receptions are nonblocking whereas they were blocking in the synchronous version. So, after the sendings, a processor takes the last version of its neighbors' messages if new messages have arrived since the previous iteration. In Algorithm 5.1, restarting a new iteration without receiving any new messages leads to the same computation. That is why a simple way to enhance this algorithm consists in detecting if a new message has arrived at each iteration. If this is not the case, it would probably be better to wait for a few micro seconds and to test again if a new message has arrived. According to the number of neighbors, it may be wise to wait for the reception of a given number of messages.

The other difference with the synchronous version of the Jacobi algorithm concerns the convergence detection. As described in Section 5.7, it is possible to use a centralized version or a decentralized version in order to detect the convergence. In this algorithm we consider that a function called *convergence* allows us to detect the global convergence. This function uses the local error and the threshold *Epsilon* used to stop the iterations.

If receptions are directly managed by threads, they can occur at any moment in the program. In that case, Algorithm 5.1 is slightly different. In fact, the call to the receive function is no longer in the main iteration. As receptions are completely free and do not only occur at the end of an iteration, this version of the Jacobi algorithm is based on what we have called *receiver-side semi-flexibility* (cf Section 3.4.3.1.2).

5.6.3 Asynchronous block Jacobi algorithm

This algorithm is the asynchronous version of the synchronous block Jacobi one. Compared to the synchronous version, it presents the advantage of being less perturbed by synchronizations as each processor has a block of the matrix to compute with a direct method that may require a non-negligible time for each subsystem. Compared to the Jacobi algorithm (without blocks), the asynchronous block version will probably converge in less iterations but they will probably be longer. So, during the solving of the subsystem, messages from the neighbors have time to arrive. Consequently, the overlapping of messages by computations with this algorithm may be more important than in the asynchronous Jacobi algorithm, especially when dealing with large matrices.

This algorithm, like its synchronous version, also requires that the linear solver used to solve the subsystem provides a result without approximation. In other words, a direct method is required.

Algorithm 5.1 Asynchronous 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
Epsilon: desired accuracy
Converged: convergence state

repeat
  for i=0 to Size−1 do
    X[i] ← 0
    for j=0 to i+Offset−1 do
      X[i] ← X[i]+A[i][j]×XOld[j]
    end for
    for j=i+Offset+1 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(A[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
  Converged ← convergence(Error, Epsilon)
until Converged = true
```

Algorithm 5.2 Asynchronous 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
Epsilon: desired accuracy
Converged: convergence state

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−1 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
  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
  Converged ← convergence(Error, Epsilon)
until Converged = true
```

5.6.4 Asynchronous multisplitting algorithm for solving linear systems

The asynchronous version of the multisplitting method for solving linear systems is designed to be efficient for grid or distant clusters. This method actually features interesting characteristics for this. It is a coarse grained algorithm since a processor solves the subsystem it is in charge of at each iteration either using a sequential iterative solver (i.e., so we obtain a two-stage algorithm) or a direct one. According to the characteristics of the subsystems obtained by the splitting and the parameters of the architecture, a good choice of the inner method can drastically change the performances. This method allows us to overlap communications with computations. This feature is typically provided by the asynchronism of the method. Consequently, we strongly believe that this method is particularly well suited to solve large linear systems in grid environments. Compared to the synchronous version, the asynchronous one only has two modifications. Those two modifications concern the two main differences between a synchronous and an asynchronous version of the same algorithm for which the convergence proof in the asynchronous mode has been previously studied, that is to say, the management of the communications and the convergence detection, as previously mentioned in this section.

With this method, it is strongly recommended to count the number of messages received per iteration and to take into account this number in order to decide if the program should wait for other messages or run the next iteration. As previously mentioned, running a new iteration without any new message will produce the same result, which is not interesting from the computational point of view. In order to increase the convergence speed it is sometimes more interesting to wait for a small span of time, for example 1 ms, to receive some new messages rather than using only one new message before running the next iteration. In the synchronous version of this algorithm we have presented the multiple ways of overlapping some components. In Algorithm 5.4 we present the small changes in the multisplitting algorithm in order to take into account the *Overlap* components which are overlapped. Obviously, we consider that the size of the *Overlap* parameter is less than the size of the subsystem.

Using the overlapping of components has two main impacts on the execution of an AIAC algorithm. The first one is that the number of iterations required to reach the convergence threshold is smaller. That is the positive point. The second impact, which is a drawback, is that the size of each subsystem is larger, and consequently, the time to solve a subsystem is longer. That is why using the overlapping mechanism may reduce the number of iterations when this number is high, i.e., the spectral radius of iteration matrix is close to one. Nevertheless, according to the method used to solve subsystems, the solving time may change. If a direct method is used, then one of the most time-consuming tasks consists in factorizing the matrix. At each iteration of the multisplitting method, only the right-hand side changes, so the factorized

Algorithm 5.3 Asynchronous 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 of 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
Epsilon: desired accuracy
Converged: convergence state

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
      if Recv(i, TLoc) then
        Update XLeft or Xright with TLoc according to the processor $i$
      end if
    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
  Converged ← convergence(Error, Epsilon)
until Converged = true
```

Algorithm 5.4 Parameter to take into account the overlapping for the multisplitting method

```
if MyRank=0 or MyRank=NbProcs−1 then
  Size ← Size+Overlap
else
  Size ← Size+2×Overlap
end if
if MyRank≠0 then
  Offset ← Offset−Overlap
end if
```

form of the matrix can be re-used for the next iterations. So, when the number of iterations to reach the convergence threshold is high, the time to factorize a matrix may not be so important in comparison to the number of times that the factorized form will be used. If an iterative method is used, the time to solve a subsystem may vary linearly with the size of a sparse matrix. Hence, it may be worth overlapping some components but it is difficult to define an optimal overlapping size. Furthermore, the optimal size may depend on the network speed, because if the bandwidth is low, it may be preferable to compute longer and communicate less.

5.6.5 Asynchronous Newton-multisplitting algorithm

In Algorithm 4.8 we have described the synchronous version of the Newton-multisplitting algorithm. In order to define the asynchronous version of that algorithm, presented in Algorithm 5.5, we can use the same variables (c.f. Algorithm 4.7), except that instead of the variable $MaxErrorMulti$ we need a boolean *Converged* as in all other AIAC algorithms. It should be noted that in the asynchronous Newton-multisplitting algorithm only one part is asynchronous, this is the computation of the solution of the linear system obtained at each iteration of the Newton process. So, the Newton iterations are still synchronous.

In Figure 5.1, we illustrate the behavior of the algorithm. At each Newton iteration, a synchronization step is used; it is represented by a vertical line in the figure. The synchronization corresponds to the computation of the global error of the Newton process and to the diffusion of the local values of components of vector X computed on each processor. Rectangles represent iterations of the multisplitting method used to solve the linear system obtained at each Newton iteration. So, this figure clearly highlights that Newton iterations are synchronized whereas multisplitting iterations are asynchronous.

Algorithm 5.5 Asynchronous 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
  Converged $\leftarrow$ false
  repeat
    FLoc $\leftarrow$ F
    if MyRank $\neq$ 0 then
      FLoc $\leftarrow$ FLoc$-$JDepLeft$\times$DXLeft
    end if
    if MyRank $\neq$ NbProcs$-$1 then
      FLoc $\leftarrow$ FLoc$-$JDepRight$\times$DXRight
    end if
    DX $\leftarrow$ Solve(J, FLoc)
    for i=0 to NbProcs$-$1 do
      if i $\neq$ MyRank and DependsOnMe[i] then
        Send(i, PartOf(DX, i))
      end if
    end for
    for i=0 to NbProcs$-$1 do
      if i $\neq$ MyRank and IDependOn[i] then
        if Recv(i, TLoc) then
          Update DXLeft or DXRight with TLoc according to processor $i$
        end if
      end if
    end for
    ErrorMulti$\leftarrow$ 0
    for i=0 to Size$-$1 do
      ErrorMulti $\leftarrow$ max(ErrorMulti, abs(DX[i]$-$DXOld[i]))
      DXOld[i]$\leftarrow$ DX[i]
    end for
    Converged $\leftarrow$ convergence(ErrorMulti, EpsilonMulti)
  until Converged = true
  X $\leftarrow$ X+DX
  ErrorNewton$\leftarrow$ 0
  for i=0 to Size$-$1 do
    ErrorNewton $\leftarrow$ max(ErrorNewton, abs(DX[i]))
  end for
  AllToAllV(X[Offset], X, Size)
  AllReduce(ErrorNewton, ErrorNewtonMax, Max)
until stopping criteria of Newton is reached
      (MaxErrorNewton $\leq$ EpsilonNewton)
```

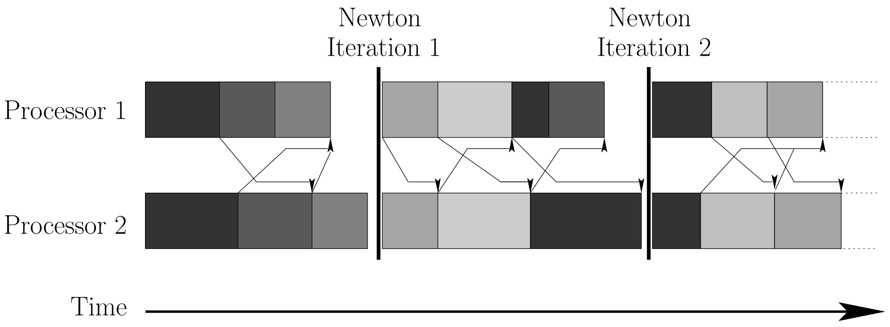

FIGURE 5.1: Iterations of the Newton-multisplitting method.

5.6.6 Asynchronous multisplitting-Newton algorithm

This algorithm allows us to solve a nonlinear system. It is formally described in Section 5.5.2. Algorithm 5.6 describes a possible implementation. In order to see the difference in terms of the computations, we represent in Figure 5.2 the decomposition of the problem. In fact, the problem is not considered in its globality like in the Newton-multisplitting method. The function F is split into $NbProcs$ processors. And each of them must solve a different part of the function F using the Newton process. So, the Jacobian matrix is not considered in its totality as in the previous algorithm. Each processor computes a local Jacobian matrix of size $Size \times Size$ which corresponds to the local size after decomposition. As in the previous algorithm the computation of the local function F requires a larger part of the vector X than the one locally computed. This is why in Algorithm 5.6 we consider that this vector has the global size of the system. In opposition to the Newton-multisplitting algorithm for which the multisplitting is used to solve a linear system obtained at each Newton iteration, the multisplitting in this algorithm is used to split the nonlinear system, i.e., the Newton method. So, this method has only one iteration in which messages are used to update the approximation of the vector X that is computed locally with Newton iterations on local subsystems.

From the programming point of view, this method is simpler to implement than the Newton-multisplitting one. Figure 5.2 allows us to understand how the decomposition is different from the previous one. It is easy to see that the parts called $JDepLeft$ and $JDepRight$ in Figure 4.12 are completely ignored. Likewise, there is no need for processors to exchange their local solutions of each subsystem (in opposition to the Newton-multisplitting method) since this is done by directly exchanging vector X. As already mentioned in the previous chapter, in all the multisplitting methods it is possible to solve the subsystems using either a direct method or an iterative one. In the latter case, we obtain a two-stage algorithm.

Algorithm 5.6 Asynchronous multisplitting-Newton 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
JLoc[Size][Size]: local block-diagonal part of the Jacobian matrix
DependsOnMe[NbProcs]: array of the dependent processors
IDependOn[NbProcs]: array of the processors this processor depends on
F[Size]: right-hand side vector of the subsystem
X[SizeGlo]: solution vector of the subsystem
DX[Size]: solution vector of the multisplitting subsystem
TLoc[Size]: array used for the receptions of the dependencies
Error: local error
Epsilon: desired accuracy
Converged: convergence state

repeat
  if first iteration or required then
    Computation of the Jacobian submatrix and storage of the result into
    JLoc
  end if
  Computation of −F depending on X from components Offset to
  Offset+size−1 and storage of the result into F
  DX ← Solve(JLoc, F)
  for i=0 to Size−1 do
    X[Offset+i] ← X[Offset+i]+DX[i]
  end for
  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
      if Recv(i, TLoc) then
        Update X according to processor $i$
      end if
    end if
  end for
  Error← 0
  for i=0 to Size−1 do
    ErrorMulti ← max(Error, abs(DX[i]))
  end for
  Converged ← convergence(Error, Epsilon)
until Converged = true
```

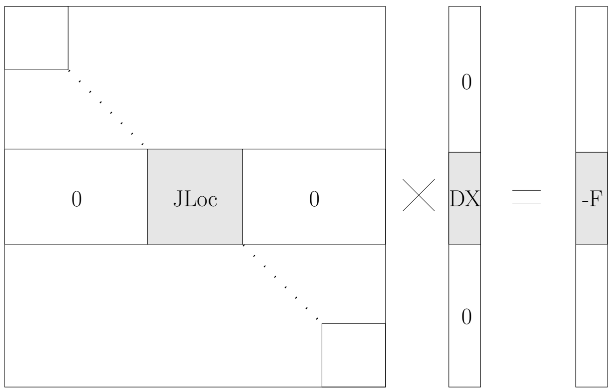

FIGURE 5.2: Decomposition of the multisplitting-Newton.

In Figure 5.3 we represent iterations of the multisplitting-Newton method. The rectangles represent the iterations of the Newton process on each processor. As can be seen, those iterations are asynchronous. So, compared to Figure 5.1 it is obvious that the multisplitting-Newton algorithm may be faster than the Newton-multisplitting one when communication delays are in favor of asynchronous iterations. This is typically the case in distant clusters in which the communication links and the machines are generally heterogeneous, implying large disparities in the communication and computation speeds in the system.

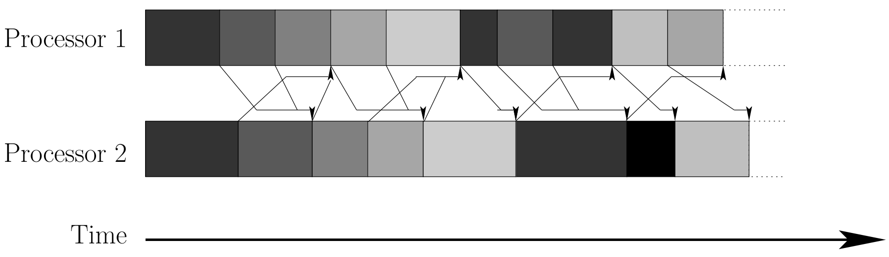

FIGURE 5.3: Iterations of the multisplitting-Newton method.

5.7 Convergence detection

As seen in Section 4.5, the convergence detection is an important issue of iterative algorithms. In the asynchronous context, the convergence detection is even hardened by the difficulty to get a correct image of the global state at any time during the process.

The most common techniques used in distributed computing to recover that information are centralized [55, 43, 100] and synchronous [84]. By their nature, those detection algorithms are efficient in parallel systems with a small physical radius but are not suited to large scale and/or distant distributed systems. Moreover, they are not suited to asynchronous iterative algorithms either, as the global synchronizations required at each recovery of the global state would indirectly synchronize the iterative process itself and then would drastically reduce the ratio of asynchronism and its benefit.

In fact, specific studies about the termination detection have been led in the context of asynchronous iterative algorithms [33, 104, 37]. But, most of them were either centralized or based on particular assumptions sometimes including some modifications of the iterative process itself.

So, in order to preserve the benefit of the asynchronism, the convergence detection algorithm must also be asynchronous. Moreover, the centralization of such an algorithm may not only generate the classical problem of bottlenecks but may also induce a loss of generality in its possible contexts of use. Indeed, in the classical centralized algorithms, all processors directly communicate their information to the central one. However, that communication scheme, implying that one machine can directly be contacted by all the others, is not possible in all parallel systems, particularly in the distributed clusters in which each site may have restricted access policies for security reasons. In most cases, only one machine of a given cluster is reachable from the outside. In order to bypass that problem, an explicit forwarding of the messages can be performed from any node in the system toward the central one. That method presents the advantage of only involving communications between neighboring nodes and is adapted to the hierarchical communication systems that can be found in distributed clusters. Unfortunately, that scheme implies more communications, slowing down the network and indirectly the iterative process itself. Moreover, it also implies larger delays toward the central node.

So, the most suitable detection algorithm in that context must not only be asynchronous but also completely decentralized. Such an algorithm is presented below.

5.7.1 Decentralized convergence detection algorithm

The decentralized algorithm for global convergence detection presented here works on all parallel iterative algorithms, either asynchronous or synchronous.

Although the version described in the following is closer to asynchronous algorithms, which represent the most general case, only a few adaptations are necessary to use it in the synchronous context.

The major difficulty with termination detection lies in the proof that the proposed algorithm does not detect convergence prematurely. Indeed, in asynchronous algorithms, the delays between iterations could lead to a false realization of the convergence criterion. This situation typically occurs in heterogeneous contexts, for example when a processor computes a new iteration whereas a slower processor computes a former iteration. That difficulty is increased with distant processors where the communication/computation ratio may be important.

As for the classical convergence detection algorithms, the principle of the decentralized detection algorithm is based on two steps. The first one consists in detecting the local convergence on each processor and the second one properly consists in the global convergence detection. Those two steps are described in the following paragraphs.

5.7.1.1 Local convergence detection

The local convergence step is quite similar to the one used in the synchronous case. As explained in Section 4.5, there is usually no information about the distance between the current state of the system and its fixed point. So, in place, the residual is used according to a chosen metric to get an idea of the stabilization of the process. Finally, that stabilization is itself determined by the setting of a threshold on the residual. However, it has also been seen in the previous chapter that when the metric used is not the contraction one, the residual does not follow a monotonous decrease but there may be oscillations around the given threshold. Hence, if no care is taken, a local convergence can be detected too early, leading in turn to a false detection of the global convergence. Once again, we insist on the fact that this problem is common to *all* iterative algorithms and is *not* due to the asynchronism.

Currently, there is no way to ensure a definitive local convergence on a processor without modifying the iterative process, as in [33]. The common heuristic is then to assume that local convergence is achieved when the node has performed a given number of successive iterations under the residual threshold. That mechanism is used in Algorithm 5.7. It implies the use of a constant, called $THRESHOLD\_LOCAL\_CV$, which represents the required number of successive iterations under the residual threshold to ensure the local convergence. It is important to note that this $THRESHOLD\_LOCAL\_CV$ value theoretically exists and is finite since, by hypothesis, the asynchronous iterative process converges. However, that value is quite difficult, not to say impossible, to evaluate in practice. Consequently, the use of an approximate value implies that the detection of the local convergence may not be definitive as the residual may rise again over the threshold after the considered number of iterations passed under it. Hence, in that context, the local state of a node may alternatively vary between convergence and non-convergence. This is why two versions of the detection algorithm are presented in the following: a theoretical version, not affected by that problem, which is useful to describe and prove the overall detection scheme, and a practical version which takes into account that problem of local states alternation.

5.7.1.2 Global convergence detection

The goal here is to obtain a similar stopping criterion as in the sequential/synchronous modes, that is to say, having all the nodes in local convergence at the same time. Unfortunately, if the asynchronism is not responsible for the difficulty in evaluating the local convergence, it hardens the global convergence detection by making the building of a representative image of the global state of the system more difficult. The process described below allows us to detect the global convergence on any one node of the system in a decentralized manner. Its correctness is proved in the context where the contraction norm is used. In other cases, often encountered in practice, the process is still correct but an additional verification step is necessary after the global detection to ensure that the system was in the correct global state at the detection time.

5.7.1.2.1 Global detection scheme: The decentralization of the detection algorithm is based upon a scheme quite similar to the leader election protocol [83]. That protocol consists in dynamically designating one processor to perform a given task. In that case, the task will be the global convergence detection. However, in that particular context, the leader election process requires some specific adaptations which imply the use of a tree graph. Fortunately, that does not reduce the generality of the algorithm since it is always possible to compute (off-line or in-line) a spanning tree from any connected graph.

The election process works with what can be called $PartialCV$ messages between processors. Such a message informs the receiver that all the processors in the subtree depending on the sender (behind the sender according to the receiver) have reached local convergence. Hence, on each processor, the algorithm considers the number of neighbors (in the tree) from which no $PartialCV$ message has already been received.

When that number is equal to one and the node is in local convergence, it sends a $PartialCV$ message to its last neighbor which has not sent it such a message yet. It is at that point that the spanning tree is necessary. It ensures that there always exists at least one node in the system which only has one neighbor (all the leaves of the spanning tree). Thus, the partial convergence detections will propagate from the leaves of the spanning tree toward the inner nodes and will meet on one node. So, as depicted in Figure 5.4, a node will detect the global convergence when it has received the $PartialCV$ messages from all its neighbors and is itself in local convergence.

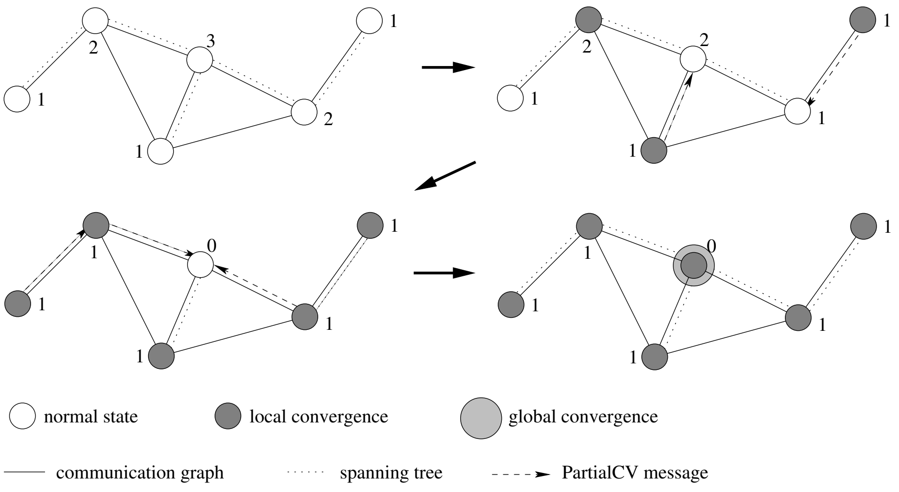

FIGURE 5.4: Decentralized global convergence detection based on the leader election protocol. For each node, the number of its neighbors in the spanning tree from which no partial convergence message has been received is indicated.

The way the process is designed implies that such a detection may happen on two neighboring nodes in place of only one. This occurs when all the nodes in the system are in local convergence and the propagation of the *PartialCV* messages ends at two neighboring nodes which are for each other the last one which has not yet sent its *PartialCV* message to the other one. So, both those nodes send their message to the other, implying a double detection of the global convergence on the two nodes. Such a particular situation is presented in Figure 5.5.

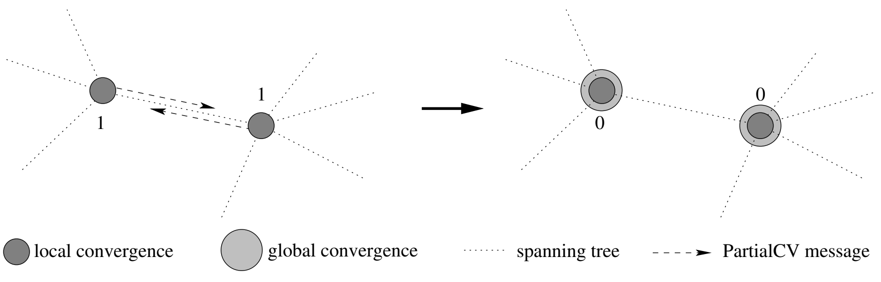

FIGURE 5.5: Simultaneous detection on two neighboring nodes.

Fortunately, that situation is not a problem per se in that context since it does not correspond to a false detection but only to a multiple one. Moreover, as the halting procedure is performed by the propagation of halting messages throughout the system from the elected node(s) and each node forwards the halting message to its other neighbors only once, that special case generates only two useless messages between the two elected nodes. So, it does not alter the halting process and does not actually require any particular treatment. However, if for some reason (often a practical one) only one node had to be elected, this could be easily achieved with, for example, a simple verification and choice mechanism between a node which detects the global convergence and its neighbor from which has come the last *PartialCV* message.

NbNeig	integer representing the number of neighbors in the spanning tree
RecvdPCV[NbNeig]	boolean array indicating for each neighbor of the current node in the spanning tree if a PartialCV message has been received from that node
NbNotRecvd	number of neighbors from which no PartialCV message has been received yet
NbUnderTh	number of successive iterations with a residual under the threshold
UnderTh	boolean equals true when the residual is under the threshold and false otherwise
LocalCV	boolean equals true when the local convergence is detected and false otherwise
GlobalCV	boolean equals true when the global convergence is detected and false otherwise

Table 5.1: Description of the variables used in Algorithm 5.7.

The decentralized detection algorithm obtained is given in Algorithm 5.7. For clarity sake, a description of the variables used in that algorithm is given in Table 5.1.

The receipts of messages are handled by distinct functions and do not directly appear in the main algorithm. That organization is particular to asynchronous algorithms where communications are not performed and managed at specific times in the algorithm but as soon as they are required or they occur.

The function *RecvPartialCV* only consists in decreasing the number of neighbors which have not yet reached local convergence. The function *recvGlobalCV* consists in stopping the iterative process on the node by setting the *GlobalCV* variable to *true*.

Algorithm 5.7 Decentralized global convergence detection

```
for all $P_i, i \in \{1, \ldots, N\}$ do
   NbNotRecvd ← NbNeig
   for Ind from 0 to NbNeig−1 do
      RecvdPCV[Ind] ← false
   end for
   NbUnderTh ← 0
   UnderTh ← false
   LocalCV ← false
   GlobalCV ← false
   repeat
      if LocalCV = false then
         ... iterative process and evaluation of UnderTh ...
         if UnderTh = true then
            NbUnderTh ← NbUnderTh + 1
            if NbUnderTh = THRESHOLD_LOCAL_CV then
               LocalCV ← true
            end if
         else
            NbUnderTh ← 0
         end if
      end if
      if LocalCV = true then
         if NbNotRecvd = 0 then
            GlobalCV ← true
         else
            if NbNotRecvd = 1 then
               Send a PartialCV message to the neighbor corresponding to the
               unique cell of RecvdPCV[] being false
            end if
         end if
      end if
   until GlobalCV = true
   Broadcast a GlobalCV message to all neighbors in the spanning tree from
   which no GlobalCV message has arrived
end for
```

Algorithm 5.8 Function RecvPartialCV()

```
Extract SrcNode from the message
SrcIndNeig ← corresponding index of SrcNode in the list of neighbors
                 of the current node (−1 if not in the list)
//the test is just a precaution since such a message should always come
//from one of the neighbors in the spanning tree
if SrcIndNeig $\geq 0$ then
   RecvdPCV[SrcIndNeig] ← true
   NbNotRecvd ← NbNotRecvd−1
end if
```

Algorithm 5.9 Function RecvGlobalCV()

```
GlobalCV ← true
```

5.7.1.2.2 Validity proof: We remind the reader that the convergence detection algorithm above is to be used with any asynchronous iterative process which converges. It is important to underline that this process does not force the convergence of any asynchronous iterative process but ensures the correct convergence detection of a converging asynchronous iterative process.

Preliminary definitions:
Let $P = \{P_1, ..., P_N\}$ be the set of the processors.
Let us define NoPCVmsg(P_i, P_j, t) between two neighboring processors P_i and P_j at time t as:

$$\text{NoPCVmsg}(P_i, P_j, t) = \begin{cases} \text{true if } P_i \text{ has not yet received a PartialCV message from } P_j \\ \text{false if } P_i \text{ has received a PartialCV message from } P_j \end{cases}$$

The detection algorithm is based on two particular properties of the processors which are the local convergence and the number of neighbors having communicated their partial convergence. Since these properties evolve during the iterative process, the set $P(t)$ of processors P_i can be written as the following partition:

$$\begin{aligned} P(t) = \quad & S_0^c(t) \cup S_1^c(t) \cup \ldots \cup S_{N-1}^c(t) \\ & \cup S_0^d(t) \cup S_1^d(t) \cup \ldots \cup S_{N-1}^d(t) \end{aligned}$$

where $S_k^e(t)$ is the set of processors having at time t:

$$\begin{aligned} &NbNotRecvd = k \\ &LocalCV = \begin{cases} \text{true if } e = c \\ \text{false if } e = d \end{cases} \end{aligned}$$

The particular presentation of $P(t)$ is only for intuitive representation of the partition.

Finally, we note $t_c(i)$ the time at which processor P_i reaches local convergence and we define $t_r(k,j)$ as the receipt time of the $PartialCV$ message on P_k from P_j and $t_m(j,k,t)$ as the communication time from P_j to P_k at time t (t is included because communication times may vary during the process).
We have then:

$$t_r(k,j) = t_c(j) + t_m(j,k,t_c(j))$$

THEOREM 5.7
If the following hypotheses are satisfied:

(H1) The communication graph used for the detection process is connected and acyclic

(H2) The asynchronous iterative process converges

(H3) Communications between neighbors are achieved in a finite time

then, there exists $t_d \in \mathbb{N}$ such that

$$\begin{array}{ll} S_0^c(t_d) \neq \emptyset & \\ |S_1^c(t_d)| \geq 0 & \\ S_k^c(t_d) = \emptyset & k \in \{2, ..., N-1\} \\ S_k^d(t_d) = \emptyset & k \in \{0, ..., N-1\} \end{array}$$

□

The second statement only appears to point out that there is no particular condition on $S_1^c(t_d)$.

The proof of Theorem 5.7 is made in two steps:

(A) we prove that $S_0^c(t_d) \neq \emptyset$ implies all the other statements of Theorem 5.7

(B) we prove that $\exists t_d \in \mathbb{N}$ such that $S_0^c(t_d) \neq \emptyset$

Part (A):

Let us define $\text{Neigh}(P_i)$ the set of physical neighbors of processor P_i. In order to get the processor P_i in $S_0^c(t)$, we must have by Algorithm 5.7:

$$\forall P_j \in \text{Neigh}(P_i),\ \text{NoPCVmsg}(P_i, P_j, t) = false$$

which implies in turn for all the P_j that

$$\forall P_k \in \text{Neigh}(P_j) \setminus \{P_i\},\ \text{NoPCVmsg}(P_j, P_k, t) = false$$

and by recursion, we deduce that

$$\forall P_a \in P(t) \setminus \{P_i\},\ \exists P_b \in P(t),\ \text{NoPCVmsg}(P_b, P_a, t) = false \qquad (5.42)$$

This means that all the P_a in that equation have sent a *PartialCV* message to the corresponding P_b and by Algorithm 5.7, this is only possible once P_a has reached local convergence.

Thus, we have:

$$\forall P_a \in P(t) \setminus \{P_i\},\ P_a \notin \bigcup_{u=0}^{N-1} S_u^d(t)$$

and since $P_i \in S_0^c(t)$, then

$$\bigcup_{u=0}^{N-1} S_u^d(t) = \emptyset$$

Moreover, by Algorithm 5.7, we also know that the condition for a processor P_a to verify Equation (5.42) (sending of a *PartialCV* message to another node) is to have its $NbNotRecvd$ equal to one.

Hence:

$$\forall P_a \in P(t) \setminus \{P_i\},\ P_a \in \bigcup_{u=0}^{1} S_u^c(t)$$

and then

$$\bigcup_{u=2}^{N-1} S_u^c(t) = \emptyset$$

and all the other statements of Theorem 5.7 are verified. □

Part (B):

By definition, at the beginning of the process, the following statements are verified:

$$\begin{array}{ll} S_k^c(0) = \emptyset & \forall k \in \{0, ..., N-1\} \\ S_0^d(0) = \emptyset & \\ S_1^d(0) \neq \emptyset & \end{array} \tag{5.43}$$

The third statement comes from (H1) which implies that the graph always has at least one node with only one neighbor.

By (H2), we have:

$$\begin{array}{c} P_i \in S_k^d(t),\ i \in \{1, ..., N\},\ k \in \{0, ..., N-1\} \\ \Rightarrow \exists\ t_c(i) \in \mathbb{N},\ \forall t \geq t_c(i), P_i \in \bigcup_{u=0}^{k} S_u^c(t) \end{array} \tag{5.44}$$

hence

$$\exists t'(k) \in \mathbb{N}, \forall t \geq t'(k),\ |S_k^d(t)| = 0,\ k \in \{0, ..., N-1\} \tag{5.45}$$

Equation (5.43), Equation (5.45) and Algorithm 5.7 imply that

$$\exists t_{dn}, \quad \begin{cases} S_1^c(t_{dn}) \neq \emptyset \\ \forall t < t_{dn}, \ \bigcup_{u=0}^{N-1} S_u^d(t) \neq \emptyset \\ \forall t \geq t_{dn}, \ \bigcup_{u=0}^{N-1} S_u^d(t) = \emptyset \\ \forall t < t_{dn}, \ S_0^c(t) = \emptyset \end{cases} \tag{5.46}$$

The last statement is, in fact, a deduction from the second one. As seen in part (A), $S_0^c(t) \neq \emptyset$ implies that $\bigcup_{u=0}^{N-1} S_u^d(t) = \emptyset$ which is in contradiction with the second statement for each $t < t_{dn}$.

Now, at t_{dn}, we know by Equation (5.46) that $S_1^c(t_{dn}) \neq \emptyset$. So, every $P_i \in S_1^c(t_{dn})$, according to Algorithm 5.7, sends a *PartialCV* message to its unique neighbor P_k which verifies $\text{NoPCVmsg}(P_i, P_k, t_{dn}) = \text{true}$.
We define:

$$A(t) = \{P_i \in S_1^c(t), \ \exists! P_k \in P(t), \\ \text{NOpCVmess}(P_i, P_k, t) = \text{NOpCVmess}(P_k, P_i, t) = \text{true}\}$$

and

$$B(t) = \{P_k \in P(t), \ \exists P_i \in A(t) \ \text{ such that } \text{NoPCVmsg}(P_i, P_k, t) = \text{true}\}$$

So, $A(t)$ is the set of processors whose sending of the *PartialCV* message to exactly one element of $B(t)$ (corresponding set of destination nodes) has not yet arrived at time t.

From (H1), we deduce the following lemma.

LEMMA 5.1
Considering the set A and time $t' \geq t_{dn}$:

$$A(t'-1) \neq \emptyset, \ A(t') = \emptyset \quad \Rightarrow \begin{cases} \forall t \geq t', A(t) = \emptyset \\ \exists P_i \in S_0^c(t') \end{cases}$$

□

Justification of Lemma 5.1:

Since $t' \geq t_{dn}$, we are in the context of Equation (5.46) where all the processors are in the subsets $S_u^c, u \in \{0, ..., N-1\}$.

If we consider the state of the system at time t', it is not possible to have one node in another subset than S_0^c or S_1^c since this would imply that this node has not yet received the *PartialCV* message from at least two of its neighbors.

So, either these neighbors are communicating their *PartialCV* message to this node, which is a contradiction to $A(t') = \emptyset$, or the other possibility is that these neighbors have not sent their *PartialCV* message to this node yet. Nevertheless, the only way for these neighbors not to have sent their *PartialCV* message to this node yet is that they have themselves at least two neighbors from which they have not received the *PartialCV* message yet. If we continue this reasoning by recursion, we come to the conclusion that this situation is only possible if all these nodes form a cycle in the graph which is a contradiction to hypothesis (H1).

Hence, we are sure that all the nodes have reached their local convergence and sent a *PartialCV* message which has already arrived at the destination node.

Finally, (H1) also implies that there is at least one node which has received the *PartialCV* messages from all its neighbors and is then located in $S_0^c(t_{dn})$.

REMARK 5.1 One consequence is that as soon as the set A becomes empty, it cannot become nonempty again. ▯

REMARK 5.2 Another consequence is that time t' is equivalent to time t_d in Theorem 5.7 since $S_0^c(t') \neq \emptyset$ and then Part (A) of the proof implies all the other statements of the theorem. ▯

REMARK 5.3 At time t_{dn}, all the processors have reached their local convergence and since $S_1^d(0) \neq \emptyset$ it is sure that the set A becomes nonempty at the latest at time t_{dn}. ▯

Now, let us examine the set $A(t_{dn})$:

If it is empty, Lemma 1 and Remark 5.3 imply that it was nonempty at the time just before and then t_{dn} corresponds to the time t' in Lemma 1 which also corresponds to the time t_d in Theorem 5.7 as pointed out by Remark 5.2.

If it is nonempty, Equation (5.46) implies that $B(t_{dn}) \subseteq \bigcup_{u=1}^{N-1} S_u^c(t_{dn})$ and there are two distinct possibilities over the set $B(t_{dn})$: (5.47)

(1) $\forall P_l \in B(t_{dn}), P_l \in S_1^c(t_{dn})$

(2) $\forall P_l \in B(t_{dn}), P_l \in \bigcup_{u=2}^{N-1} S_u^c(t_{dn})$

Case (1):

In this case, there exists at least one $P_l \in B(t_{dn})$ such that $\exists! P_i \in A(t_{dn})$ for which $\text{NoPCVmsg}(P_l, P_i, t_{dn}) = \text{true}$ and $\text{NoPCVmsg}(P_l, P_i, t_r(l,i)) = \text{false}$ implying $P_l \in S_0^c(t_r(l,i))$, and leading to the detection of the global convergence on P_l at time $t_r(l,i)$. Hypothesis (H3) ensures that $t_r(l,i) < \infty$ and then statement (B) is verified with $t_d = t_r(l,i)$.

Case (2):

In this case, $P_l \in B(t_{dn})$ implies that there is one $S_u^c(t_{dn})$, $u \in \{2, ..., N-1\}$ such that $P_l \in S_u^c(t_{dn})$, and then by Algorithm 5.7:

$$P_l \in \bigcup_{v=0}^{u-1} S_v^c(t_r(l,i)), \quad \text{with } i \text{ such that } P_i \in A(t_{dn}) \\ \text{and NoPCVmsg}(P_i, P_l, t_{dn}) = \text{true} \tag{5.48}$$

This means that each time a processor receives a *PartialCV* message, its number of neighbors which have not sent it a *PartialCV* message yet decreases by one. Moreover, we use a union of the $u-1$ first subsets because this processor may receive other *PartialCV* messages from other neighbors in the interval time between t_{dn} and $t_r(l,i)$, making it move down by more than one subset.

Hence, by Lemma 1:

- either there exists at least one $P_l \in B(t_{dn})$ for which $P_l \in A(t_r(l,i))$, with $t_r(l,i) < \infty$ by (H3), and we come back to a similar context as in (5.47) where $A(t) \neq \emptyset$ by replacing t_{dn} by $t_r(l,i)$ and we obtain a recursion on $\bigcup_{u=1}^{N-1} S_u^c(t)$. Equation (5.48) ensures that this recursion will empty all the subsets $S_u^c(t), u \in \{2, ..., N-1\}$ and will then converge toward case (1).
- or none of the nodes of $B(t_{dn})$ comes in the set A which becomes empty as soon as all the nodes of $B(t_{dn})$ have received their *PartialCV* message (in a finite time by (H3)), directly leading to Theorem 5.7 by Remark 5.2.

□

As a last remark, it can be noticed that hypothesis (H3) also implies that the termination of the iterative process on all nodes happens in a finite time after the global convergence detection on the elected node (at time t_d in Theorem 5.7).

5.7.1.2.3 Practical version: As mentioned at the beginning of Section 5.7.1.2, Algorithm 5.7 is only usable in that form when the contraction metric is known and used to compute the residual. When that metric is unknown, the difficulty of ensuring the local convergence on each node implies the use of two additional mechanisms: an optional one which is useful to regulate the local detections better, and a vital one which permits us to get a correct image of the global state of the system. Indeed, the possible alternation of the local state of the nodes requires a more accurate snapshot of the global state of the system to ensure that all the nodes have verified the local convergence conditions at the same time.

The practical version presented here is somewhat different from the one proposed in [17]. That previous version has the drawback of requiring the determination of the maximal communication time between any couple of nodes in the system during the entire iterative process. In practice, it is quite difficult, if not to say impossible, to get an accurate estimation of such a value. The version described here does not use that value and, more generally, presents the advantage of not requiring any specific information on the parallel system used. Its approach is closer to the theoretical version presented in the previous part in the sense that it lets the global detection happen even if the local evolutions on the nodes change during the election process. Then, after the global detection, it includes an additional verification phase to ensure its validity. It is important to notice here that the iterative process is not interrupted either during the global convergence detection process or during the verification phase. There are two reasons for that; the most obvious one is not to slow down the iterative process itself and the second one is that its evolution during the global detection and verification processes represents a mandatory piece of information.

Concerning the first of the two mechanisms mentioned above, it concerns the local convergence detection on each node and consists in taking into account what we call pseudo-periods in place of a given number (arbitrary in practice) of successive iterations. In the domain of dynamic systems, a period corresponds to a minimal span of time during which all the components of the system are updated at least once with different data values from its dependencies. The pseudo-period is quite a local version of that global progression step. So, for each node, a pseudo-period corresponds to the minimal span of time during which that node receives at least one newer data message from all its dependencies. In this way, the local evolution of one node is fully representative between two consecutive pseudo-periods. Thus, the local convergence detection is no longer assumed after a given number of successive iterations with the residual under the threshold but after at least one (but possibly several successive ones) pseudo-period verifying that constraint. This has a drastic regulating effect on the local convergence detections in practice and, if it cannot avoid all the false detections due to an inadequate used norm, it sharply limits them. It is therefore strongly recommended although not essential.

For its part, the second mechanism is imperative and takes place at the global level of the system just after the global convergence detection. Its aim is to verify that all the nodes were still in local convergence at the time of the global detection and that their states were representative of their evolution. Hence, that verification is decomposed in four main steps:

1) Diffusion of a verification message from the elected node through the spanning tree to initiate the verification phase.

2) Elaboration on each node of its response to the verification request.

3) Gathering of the responses of all the nodes toward the elected node through the spanning tree to get the verdict.

4) Diffusion of a verdict message from the elected node through the spanning tree to finish the verification phase.

When one node is elected by the global convergence detection process, it sends a verification message to all its neighbors (step 1). Each node which receives such a message from one of its neighbors (referred to as the asking node in the following) forwards it to all its other neighbors in the spanning tree (step 1) and, while waiting for their responses, elaborates its own response (step 2). The response of a node does not only depend on its own state and evolution but also on the responses of its neighbors in the spanning tree, except the asking one. As soon as the response is available, the node returns it to its asking node (step 3). Finally, when the elected node has its own response and those of its neighbors, it deduces the verdict and sends it to all its neighbors (step 4). Then, each node receiving a verdict message forwards it to its other neighbors in the spanning tree (step 4). At the end of the verification phase, the state of each node is set up according to the final verdict. The global scheme of that verification mechanism is depicted in Figure 5.6.

As mentioned above, the response of each node depends on its state but also on its evolution during the verification phase. Effectively, in order to ensure that all the nodes have been in local convergence at the same time (which is the criterion used in the sequential and synchronous versions), the response of a node is positive if and only if its residual never goes back over the threshold during the span of time between its last sending of a $PartialCV$ message and the sending of its response to the verification request.

Moreover, to be sure that the response of each node is representative of its actual state and is not illusory, a particular mechanism is inserted to ensure that each node actually evolves during the span of time between the receipt of the verification request and the sending of its response. That mechanism roughly corresponds to the waiting of a particular pseudo-period. The ideal way to ensure the pertinence of the global state image would be to wait for a period and watch the resulting state. However, periods are quite difficult and expensive to identify in dynamical systems implemented on distributed environments. So, a lighter concept is used here which is better suited to the decentralization constraint while giving pertinent information about the evolution of the system as well. Hence, each node sends its response (depending on its residual evolution) only after having performed at least one iteration with versions of all its data dependencies at least as recent as the global detection time. In this way, the response will be fully representative of the actual evolution and state of that node until that time.

In order to force the nodes to use specific data versions during the verification phase, a tagging system is included in the data messages in order to differentiate them between the successive phases of the iterative process (normal processing and verification phase). Moreover, since there may be several verification phases during the whole iterative process, due to possible cancellations of global detections, that tagging is also useful to distinguish the data

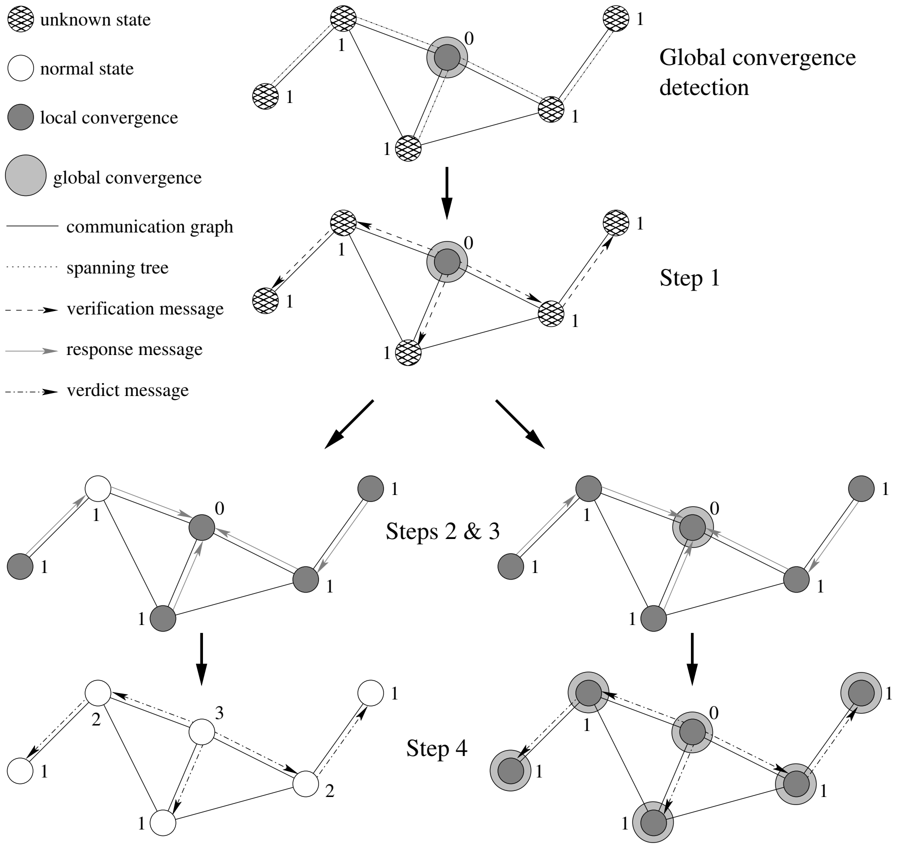

FIGURE 5.6: Verification mechanism of the global convergence. The two possibilities are illustrated, cancellation on the left and confirmation on the right. The unknown states in the first two steps are from the elected node point of view. The value beside each node corresponds to the variable $NbNotRecvd$.

messages related to the different verification phases. Finally, there is another good reason to use tagged messages not only for the data communications but also for the information related to the global detection and verification processes: the reactivity of the verification phase.

In fact, as quite an important number of verification phases is likely to occur during an entire iterative process, it is rather important to increase its reactivity. This has the indirect effect of reducing the latency between the actual global convergence of the system and its detection. In order to do so, each node is allowed to send its response as soon as it is able to deduce it, that is to say, when its residual goes over the threshold or when it receives a negative response from one of its neighbors. Such events imply a negative response of the node, whatever the values of the other elements constituting its response are. It is then a waste of time to wait for the responses of the other neighbors or the local completion of a pseudo-period. The same behavior takes place on the elected node except that it directly sends a negative verdict message to its neighbors instead of sending a response to an asking node. However, as the order of the messages is not ensured in the asynchronous computing context, this strategy also implies that messages related to a given verification phase may arrive on a node after the termination of that phase (in the case of a verification phase canceled faster). Thus, in order to avoid confusions in the messages related to the global detection scheme, a tagging system must also be inserted.

Finally, in order to respond to all those message distinction constraints, each phase of the iterative process (normal computing and verification of the global convergence) is distinguished in time by an integer tag incremented at each phase transition, as shown in Figure 5.7, with four nodes linearly organized for a span of time beginning with the tag equal to k.

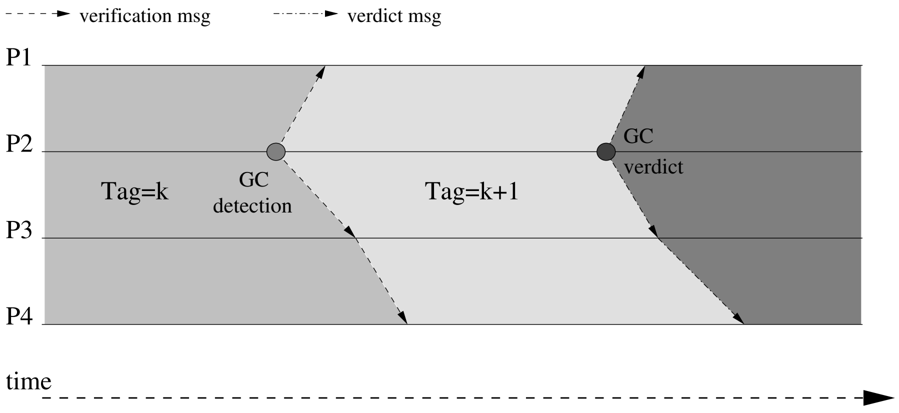

FIGURE 5.7: Distinction of the successive phases during the iterative process.

The whole detection and verification mechanism is detailed in Figure 5.8 in the same computing context as above and in the case of a global convergence detected and confirmed on node P_2. As can be seen, the whole process ensures, in case of a positive verdict, that all the nodes in the distributed system have had their residual under the threshold at least at the time at which the global convergence was detected on the elected node, and possibly during a larger span of time after it. Moreover, the pseudo-period performed on each node during the verification phase with data as recent as the global detection ensures that their states are representative of their actual evolutions. In this way, that entire global convergence detection mechanism provides a similar stopping criterion as in the sequential/synchronous cases.

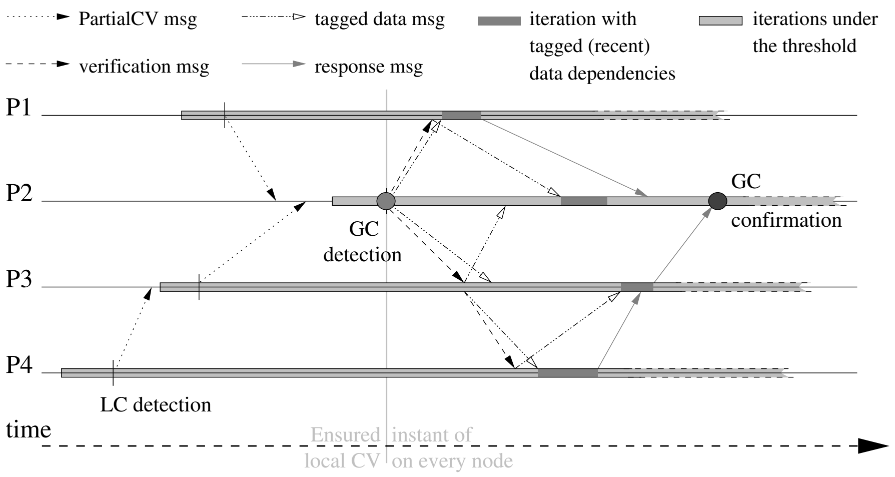

FIGURE 5.8: Mechanism ensuring that all the nodes are in representative stabilization at least at the time of global convergence detection.

As the behavior of the nodes is not the same according to the different steps in the detection process and verification phase, it is also necessary to introduce four different states:

- **NORMAL:** the basic state during the whole iterative process when the node is not in the global convergence detection mechanism.
- **WAIT4V:** when the node is waiting for the local start of the verification phase after its sending of a PartialCV message.
- **VERIF:** when the node is performing the verification phase, either after the receipt of the corresponding message or by election.
- **FINISHED:** when the global convergence has been confirmed.

The transitions between those states are depicted in Figure 5.9.

The final scheme obtained is given in Algorithm 5.15. In order to get an easy reading of it, the list of the additional variables according to the previous algorithm is given in Table 5.2.

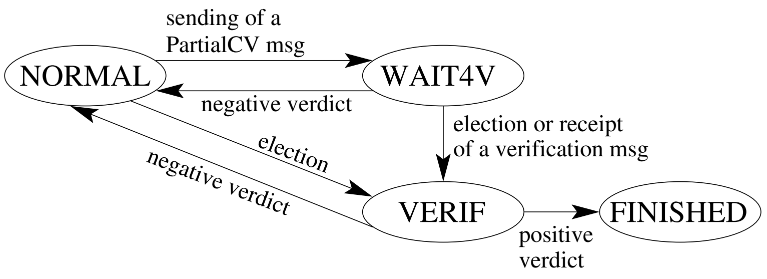

FIGURE 5.9: State transitions in the global convergence detection process.

The different types of messages are listed below together with their contents:

- **data message:**
  - identifier of the source node
  - source node iteration number at the sending time
  - source node phase tag at the sending time
  - data
- **PartialCV message:**
  - identifier of the source node
  - source node phase tag at the sending time
- **verification message:**
  - identifier of the source node
  - source node phase tag at the sending time
- **response message:**
  - identifier of the source node
  - source node phase tag at the sending time
  - response of the source node
- **verdict message:**
  - identifier of the source node
  - new phase tag to use on the receiver
  - verdict

The algorithm also uses additional functions which are briefly described below:

- **InitializeState():** (re-)initializes the variables related to the convergence detection process and sets the node in NORMAL state.
- **ReinitializePPer():** (re-)initializes the variables related to the pseudo-period detection.

MyRank	integer identifying uniquely the current node
State	integer indicating the current state of the node among NORMAL, WAIT4V, VERIF and FINISHED
PhaseTag	integer identifying the current phase on the current node
PseudoPerBeg	boolean indicating that a pseudo-period has begun
PseudoPerEnd	boolean indicating the end of a pseudo-period
NbDep	integer representing the number of computational dependencies of the current node
NewerDep[NbDep]	boolean array indicating for each data dependency if a newer version has been received since the last pseudo-period
LastIter[NbDep]	integer array indicating for each dependency node the iteration of production of the last data received from that node
PartialCVSent	boolean indicating that a PartialCV message has been sent
ElectedNode	boolean indicating that the node is the elected one
Resps[NbNeig]	integer array containing the responses of the neighbors of the current node in the spanning tree. The values are either -1 (negative), 0 (no response yet) or 1 (positive)
ResponseSent	boolean indicating that the response has been sent

Table 5.2: Description of the additional variables used in Algorithm 5.15.

- **InitializeVerif():** initializes the verification phase. In particular, the $PhaseTag$ variable is incremented to distinguish the new verification phase from the potential previous ones.

- **RecvDataDependency():** manages the receipts of data dependencies. In the general asynchronous model, each received datum is taken into account, whenever it was produced. However, taking only newer data (produced after the locally available ones) tends in practice to speed up the iterative process. So, the function takes into account any newer data when the receiver is not in verification phase (VERIF state). Otherwise, it filters the data produced after the last global convergence detection, that is to say, with the same phase tag as the receiver.

- **RecvPartialCV():** manages the receipts of PartialCV messages. Also updates the local state of the node when an election is possible. However, the mutual exclusion mechanism mentioned on page 149 is performed to ensure that only one node is elected in the system.

- **RecvVerification():** manages the receipts of verification messages. The message is taken into account only when its phase tag corresponds to the following phase on the receiver. In that case, the state of the receiver is changed to enter the verification phase (VERIF state) and the message is propagated to its other neighbors in the spanning tree.

- **RecvResponse():** manages the receipts of response messages. The message is taken into account only when the phase tag in the message corresponds to the current phase tag on the receiver.

- **RecvVerdict():** manages the receipt of the verdict of the verification phase on the non-elected nodes. The verdict is always taken into account and propagated through the spanning tree to set all the nodes either in FINISHED state or back in NORMAL state with a new phase tag. As the state of the non-elected nodes cannot change before the receipt of that message, no other global convergence detection may happen before all the nodes have received it. Therefore, there cannot be any confusion with a similar message coming from a previous verification phase.

- **ChooseLeader(integer, integer):** takes two integer parameters identifying two nodes which are potential candidates to the leader election and returns the one which is chosen by the election referee policy.

The last function of the list is not detailed in the following since it directly depends on the referee policy used. The choice of that policy is quite free as its only constraint is to make a choice between the two proposed nodes.

Algorithm 5.10 Function InitializeState()

```
NbNotRecvd ← NbNeig
for Ind from 0 to NbNeig−1 do
  RecvdPCV[Ind] ← false
end for
ElectedNode ← false
LocalCV ← false
PartialCVSent ← false
ReinitializePPer()
State ← NORMAL
```

Algorithm 5.11 Function ReinitializePPer()

```
PseudoPerBeg ← false
PseudoPerEnd ← false
for Ind from 0 to NbDep−1 do
  NewerDep[Ind] ← false
end for
```

Algorithm 5.12 Function InitializeVerif()

```
ReinitializePPer()
PhaseTag ← PhaseTag + 1
for Ind from 0 to NbNeig−1 do
  Resps[Ind] ← 0
end for
ResponseSent ← false
```

Algorithm 5.13 Function RecvDataDependency()

```
Extract SrcNode, SrcIter and SrcTag from the message
SrcIndDep ← corresponding index of SrcNode in the list of dependencies
            of the current node (−1 if not in the list)
if SrcIndDep ≥ 0 then
  if LastIter[SrcIndDep] < SrcIter
    and (State ≠ VERIF or SrcTag = PhaseTag) then
    Put the data in the message at their corresponding place according to
    SrcIndDep in the local data array used for the computations
    LastIter[SrcIndDep] ← SrcIter
    NewerDep[SrcIndDep] ← true
  end if
end if
```

Algorithm 5.14 Function RecvPartialCV()

```
Extract SrcNode and SrcTag from the message
SrcIndNeig ← corresponding index of SrcNode in the list of neighbors
             of the current node (−1 if not in the list)
if SrcIndNeig ≥ 0 and SrcTag = PhaseTag then
  RecvdPCV[SrcIndNeig] ← true
  NbNotRecvd ← NbNotRecvd−1
  if NbNotRecvd = 0 and PartialCVSent = true
    and ChooseLeader(MyRank, SrcNode) = MyRank then
    ElectedNode ← true
    InitializeVerif()
    Broadcast a verification message to all its neighbors
    State ← VERIF
  end if
end if
```

Algorithm 5.15 Practical version of Algorithm 5.7 (1/3)

```
for all  $P_i, i \in \{1, \dots, N\}$ do
    InitializeState()
    UnderTh ← false
    PhaseTag ← 0
    repeat
        ... iterative process, data sendings and evaluation of UnderTh ...
        if State = NORMAL then
            if UnderTh = false then
                ReinitializePPer()
            else
                if PseudoPerBeg = false then
                    PseudoPerBeg ← true
                else
                    if PseudoPerEnd = true then
                        LocalCV ← true
                        if NbNotRecvd = 0 then
                            ElectedNode ← true
                            InitializeVerif()
                            Broadcast a verification message to all its neighbors
                            State ← VERIF
                        else
                            if NbNotRecvd = 1 then
                                Send a PartialCV message to the neighbor corresponding
                                to the unique cell of RecvdPCV[] being false
                                PartialCVSent ← true
                                State ← WAIT4V
                            end if
                        end if
                    else
                        if all the cells of NewerDep[] are true then
                            PseudoPerEnd ← true
                        end if
                    end if
                end if
            end if
        else if State = WAIT4V then
            see that part on page 167...
        else if State = VERIF then
            see that part on pages 167 and 168...
        end if
    until State = FINISHED
end for
```

Algorithm 5.15 bis Practical version of Algorithm 5.7 (2/3)

```
for all $P_i, i \in \{1, \ldots, N\}$ do
  see that part on page 166...
  repeat
    ... iterative process, data sendings and evaluation of UnderTh ...
    if State = NORMAL then
      see that part on page 166...
    else if State = WAIT4V then
      if UnderTh = false then
        LocalCV ← false
      end if
    else if State = VERIF then
      if ElectedNode = true then
        if UnderTh = false or LocalCV = false
          or at least one cell of Resps[] is negative then
          PhaseTag ← PhaseTag + 1
          Broadcast a negative verdict message to all its neighbors
          InitializeState()
        else
          if PseudoPerEnd = true then
            if there are no more 0 in Resps[] then
              if all the cells of Resps[] are positive then
                Broadcast a positive verdict message to all its neighbors
                State ← FINISHED
              else
                PhaseTag ← PhaseTag + 1
                Broadcast a negative verdict message to all its neighbors
                InitializeState()
              end if
            end if
          else
            if all the cells of NewerDep[] are true then
              PseudoPerEnd ← true
            end if
          end if
        end if
      else
        see that part on page 168...
      end if
    end if
  until State = FINISHED
end for
```

Algorithm 5.15 ter Practical version of Algorithm 5.7 (3/3)

```
for all $P_i, i \in \{1, \ldots, N\}$ do
  see that part on page 166...
  repeat
    ... iterative process, data sendings and evaluation of UnderTh ...
    if State = NORMAL then
      see that part on page 166...
    else if State = WAIT4V then
      see that part on page 167...
    else if State = VERIF then
      if ElectedNode = true then
        see that part on page 167...
      else
        if ResponseSent = false then
          if UnderTh = false or LocalCV = false
            or at least one cell of Resps[] is negative then
            Send a negative response to the asking neighbor
            //by construction, that is the neighbor to which has been sent
            //the last PartialCV message $\Leftrightarrow$ false cell of RecvdPCV[]
            ResponseSent $\leftarrow$ true
          else
            if PseudoPerEnd = true then
              if there remains only one 0 in Resps[] then
                //that last 0 is located in the cell of the asking neighbor
                if the other cells of Resps[] are all positive then
                  Send a positive response to the asking neighbor
                else
                  Send a negative response to the asking neighbor
                end if
                ResponseSent $\leftarrow$ true
              end if
            else
              if all the cells of NewerDep[] are true then
                PseudoPerEnd $\leftarrow$ true
              end if
            end if
          end if
        end if
      end if
    end if
  until State = FINISHED
end for
```

Algorithm 5.16 Function RecvVerification()

```
Extract SrcNode and SrcTag from the message
if SrcTag = PhaseTag + 1 then
    InitializeVerif()
    State ← VERIF
    Broadcast the verification message to all its neighbors but SrcNode
end if
```

Algorithm 5.17 Function RecvResponse()

```
Extract SrcNode, SrcTag and SrcResp from the message
SrcIndNeig ← corresponding index of SrcNode in the list of neighbors
              of the current node (−1 if not in the list)
if SrcIndNeig ≥ 0 and PhaseTag = SrcTag then
    Resps[SrcIndNeig] ← SrcResp
end if
```

Algorithm 5.18 Function RecvVerdict()

```
Extract SrcNode, SrcTag and SrcVerdict from the message
if SrcVerdict is positive then
    State ← FINISHED
else
    InitializeState()
    PhaseTag ← SrcTag
end if
Broadcast the verdict message to all its neighbors but SrcNode
```

5.8 Exercises

1. Consider a linear system

$$Ax = b,$$

where A is an M-matrix and ε is a positive scalar. Prove that asynchronous algorithms associated to the Jacobi algorithm for solving this linear system converge.

2. Consider a linear system

$$(A + \varepsilon)\, x = b,$$

   where A is an M-matrix. Prove that for any $\varepsilon > 0$, asynchronous algorithms associated to the Jacobi algorithm for solving this linear system converge.

3. Consider a linear system $Ax = b$. Prove that iterative algorithms arising from the splitting of A converge asynchronously if A is monotone and the splittings are weak regular.

4. Give examples of large linear systems involving monotone matrices A and by considering weak regular splitting of A, write programs of parallel multisplitting iterative algorithms that converge asynchronously to the solution of $Ax = b$ (b given).

5. Consider the gradient iterations

$$x^{(k+1)} = x^{(k)} - \gamma A x^{(k)}.$$

   Prove that if A is diagonally dominant for a sufficiently small γ, then the gradient algorithm converges asynchronously to the solution of

$$\min \frac{1}{2} x^T A x.$$

6. Consider the 2 dimensional Dirichlet problem

$$\begin{array}{l} -\Delta u = f \text{ on } \Omega =]0,1[\, \times\,]0,1[\\ u = 0 \text{ on the boundary } \partial\Omega of \Omega. \end{array}$$

   (a) By using the finite difference method to approximate the second derivatives and following the illustration example of Chapter 1, show that the approximate solution is the solution of a linear system

$$Ax = b$$

   where A is a block tridiagonal and where each block is also a tridiagonal matrix of the form

$$\begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}.$$

   (b) Propose a convergent asynchronous algorithm to solve the approximate solution of the Dirichlet problem.

(c) Propose a parallel block convergent asynchronous algorithm to solve the approximate solution of the Dirichlet problem.

7. Consider the ordinary differential equation

$$\begin{cases} \frac{dx}{dt} = f(x,t) \\ x(0) = x_0 \\ t \in [0,T]. \end{cases} \tag{5.49}$$

Let's denote by C^1 the space of continuous functions defined on $[0,T]$ with values in $\mathbb{R}^n$. Then, we suppose that the unknown function $x \in C^1$ and that f is a continuous function.

(a) Prove that the following mapping is a norm on C^1, for $x = (x_1, ..., x_n)$,

$$n(x) = \|x\|_\infty = \max_{1\le i\le n} \max_{0\le t\le T} \|x_i(t)\|$$

(b) We suppose that f is Lipschitz continuous with respect to x, with constant L, i.e.,

$$\|f(x,t) - f(y,t)\|_\infty \le L \|x - y\|_\infty$$

and consider the following fixed point mapping

$$T(x) = y \Leftrightarrow \frac{dy_i}{dt} = f(x_1, ..., x_n, t),\ y_i(0) = (x_0)_i\,.$$

Let K be a real number such that

$$\frac{1 - e^{-KT}}{K} < \frac{1}{L}$$

Prove that T is contractive with respect to the norm $\|x\|_K = \max_{1\le i\le n} \max_{0\le t\le T} e^{-Kt} \|x_i(t)\|$.

(c) Propose a parallel asynchronous algorithm which converges to the solution of (5.49).

(d) Following Section 5.4.3 of Chapter 5, build a parallel asynchronous multisplitting algorithm to solve (5.49).

8. Implement a centralized detection convergence procedure and a decentralized one. Then compare the performances on algorithms presented in this chapter.

9. With AIAC multisplitting algorithms (linear or not), compare the behavior of versions using different solvers for solving linear subsystems. It is interesting to compare the behavior of iterative solvers against direct ones. Nevertheless, the comparison between a simple iterative solver and a more complex one, like GMRES, is also instructive. Try to point out cases where simple iterative solvers perform faster than more complex ones. Try to explain when this situation is possible.

10. Try to implement the tips described in Section 5.6.4 which consist in waiting for some new messages to arrive before running the next iterations with multisplitting algorithms for solving linear systems. According to the number of neighbors of each processor try to define an appropriate number of messages to wait for before running the next iterations.

11. Try to implement the same mechanisms as in the previous exercise with nonlinear multisplitting algorithms.

12. Compare the behavior of the Newton-multisplitting algorithm and the multisplitting-Newton algorithm for some nonlinear problems. Try to point out the threshold for which one of those algorithms seems better than the other in a distant cluster context.