A Decentralized Convergence Detection Algorithm for Asynchronous Parallel Iterative Algorithms

Jacques M. Bahi, *Member*, *IEEE*, Sylvain Contassot-Vivier, *Member*, *IEEE*, Raphaël Couturier, *Member*, *IEEE*, and Flavien Vernier

Abstract—We introduce a theoretical algorithm and its practical version to perform a decentralized detection of the global convergence of parallel asynchronous iterative algorithms. We prove that, even if the algorithm is completely decentralized, the detection of global convergence is achieved on one processor under the classical conditions. The proposed algorithm is very useful in the context of grid computing in which the processors are distributed and in which detecting the convergence on a master processor may be penalizing or even impossible as in Peer to Peer computation frameworks. Finally, the efficiency of the practical algorithm is illustrated in a typical experiment.

Index Terms—Parallel iterative algorithms, asynchronism, convergence detection.

1 Introduction

In the context of scientific computations, iterative algorithms are very well suited to a large class of problems and are in many cases either preferred to direct methods or even sometimes the single way to solve a problem. Direct algorithms give the exact solution of a problem within a finite number of operations, whereas iterative algorithms provide an approximation of it, we say that they converge (asymptotically) toward this solution. When dealing with very great dimension problems, iterative algorithms are preferred, especially if they give a good approximation in a little number of iterations [1].

Those last properties have led to a good expansion of parallel iterative algorithms. Nevertheless, most of these parallel versions are synchronous. We have shown in [2] all the interest of using asynchronism in such parallel iterative algorithms especially in a global context of grid computing. The asynchronous algorithm proposed in this work used a centralized method to detect the global convergence which was not best suited to this context.

In this paper, we propose a new noncentralized global convergence detection algorithm based only on local information of the nodes. A theoretical version designed for totally asynchronous algorithms is presented and proved. Then, a practical version working on partially asynchronous algorithms, i.e., asynchronous algorithms with bounded delays is deduced. Since this detection algorithm only works well on acyclic graphs, when the physical graph of the system contains cycles, the use of a spanning tree of the graph is sufficient to manage all the communications related to the convergence detection.

In this study, we focus on all aspects of convergence detection, from theoretical point of view up to implementation problems. The main features of our method are the following: We do not modify the iterative process but propose an additional mechanism which is superimposed onto the asynchronous iterations. Since we do not merge messages for the computation and messages for the convergence detection, our method is tolerant to the loss of computational messages. Moreover, from a practical point of view, our mechanism detects termination only a short time after the actual convergence and it uses quite a few communications as opposed to the Savari and Bertsekas algorithm which requires a lot of request messages.

The following section presents the previous studies related to termination detection and leader election algorithms. Then, the principle of asynchronous iterative algorithms are briefly described and replaced in the global context of parallel iterative algorithms. The theoretical decentralized algorithm for global convergence detection is detailed in Section 4 and the proof that this method actually detects the global convergence on at least one processor in a finite time after the global convergence has happened is given in Section 5.3. Since this theoretical algorithm is not directly applicable in all real cases, we propose a practical version of this algorithm and define its domain of validity in Section 6. The relative efficiency of this algorithm is discussed in Section 7 according to experimental results.

2 Related Works

Distributed termination detection consists in finding the time at which a distributed computation has finished. This problem has already been studied in the general context of distributed computing [3], [4], [5]. Nevertheless, most of

• *The authors are with LIFC (Laboratoire d'Informatique de l'Université de Franche-Comté), FRE CNRS 2661, IUT de Belfort-Montbéliard, BP 527, 90016 Belfort, France.*
E-mail: {bahi, scontass, couturie, vernier}@iut-bm.univ-fcomte.fr.

Manuscript received 23 Jan. 2004; revised 17 June 2004; accepted 23 June 2004; published online 23 Nov. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0023-0104.

them are based on centralized algorithms which are not suited to large scale and/or distant distributed computations. Indeed, these algorithms use the Master/Slave paradigm which is convenient for parallel computers but not for distributed parallel machines. Some studies have also been done in the synchronous context [6]. Concerning the asynchronous iterative algorithms, distributed termination detection was firstly introduced in [7] under particular assumptions, such as the particular behavior of the nodes which have reached local convergence. Moreover, Savari and Bertsekas proposed another distributed version in [8], but still under more restrictive hypotheses than the ones we use, such as FIFO communications and with modifications of the iterative process itself in order to make it terminate in finite time. In [9], El Baz also proposed a review of different methods of terminating asynchronous iterative algorithms on message passing systems. Other authors have studied implementations of asynchronous algorithms, but always with centralized convergence detection [10].

The originality of our approach is to use a leader election algorithm to manage the termination of a particular class of parallel applications in a decentralized way. In fact, it is used to perform the global convergence detection of asynchronous iterative algorithms using only local information of the nodes. The major difficulty comes from the asynchronism in the application which induces a far more complex behavior toward the convergence and then may imply false detections or delayed ones. In the literature about leader election, numerous variants are presented. The version we used in this paper has been generally used in communication protocols such as the IEEE 1394 (firewire) protocol [11] or in distributed systems [12], [13]. Some studies have been done in asynchronous systems as in [14], [15] but in the largely different context of fault tolerant networks (node crashes or communication failures) and not in order to control an application.

To the best of our knowledge, this is the first study using an election algorithm to manage the termination of asynchronous iterative algorithms.

3 What Are Parallel Iterative Algorithms?

3.1 Iterative Algorithms

Iterative algorithms have the structure

$$x^{k+1} = g(x^k), \qquad k = 0, 1, \ldots \quad \text{with } x^0 \text{ given}, \qquad (1)$$

where each x^k is an n-dimensional vector, and g is some function from $\mathbb{R}^n$ into itself. A fixed point x^* of g is characterized by the property $g(x^*) = x^*$. The goal of the iterative algorithm is to reach such a fixed point starting from any initial vector x^0.

3.2 Parallel Iterative Algorithms

The parallel version of the iterative algorithms presented above is obtained by the classical block-decomposition (see, for example, [16]).

There are two main reasons to design a decentralized convergence detection algorithm for parallel iterative algorithms. The first one is that the most general class of parallel iterative algorithms corresponds to the asynchronous iterative algorithms which are not centralized by nature. The second one is of pratical interest since, in numerous contexts of use of these algorithms, the centralization is simply not possible. The reader should refer to [16] for the theoretical formulation of asynchronous iterative algorithms. From this context, we only recall the notion of delay between data dependencies which is useful in the following of the paper. The delay of block (or processor) j according to block i is noted $d_j^i(t)$ and the data version of block j available at time t on processor i is commonly noted as $r_j^i(t) = t - d_j^i(t)$. In totally asynchronous iterations, some classical conditions are assumed over the $r_j^i(t)$ in order to ensure that the process actually iterates and then evolves (see again [16]).

3.3 AIAC Algorithms

AIAC algorithms which have been introduced in [17] are a variant of the totally asynchronous algorithms. The acronym stands for Asynchronous Iterations Asynchronous Communications. In this context, all the processors perform their iterations without taking care of the progress of the other processors. They do not wait for predetermined data to become available from other processors but they keep on computing, trying to solve the given problem with whatever data happen to be available at that time. In the literature about the domain, there is one algorithmic model corresponding to these algorithms with two different formulations, those of Bertsekas and Tsitsiklis [16] and those of El Tarazi [18]. Nevertheless, several variants can be deduced from these models depending on when the communications are performed and when the received data are incorporated in the computations, see, e.g., [19], [20]. These algorithms give very good results in the global context of grid computing as has been shown in [2]. Nevertheless, the global convergence detection algorithm used in our previous studies was centralized, which was not well suited to the context of grid computing where all the nodes may not be directly accessible to each other for security reasons. This is why we propose in this paper a decentralized algorithm for global convergence detection.

4 Decentralized Algorithm for Global Convergence Detection

We propose in this section a decentralized algorithm for global convergence detection which 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.

In the following, we consider to have a set of N processors numbered from 1 to N.

The major difficulty lies in the fact of showing 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 occurs in heterogenous contexts, for example, when a fast processor computes a new iteration whereas a slow processor computes a former iteration. This difficulty is increased with distant processors where the communication/computation ratio may be important.

The principle of our convergence detection is based on two steps. The first one consists in detecting the local

convergence on each processor and the second one is properly the global convergence detection.

4.1 Local Convergence Detection

Generally, due to limited numerical representations and asymptotic convergences, the actual values of the fixed point components cannot be exactly reached. Hence, a threshold is used to determine a sufficient accuracy near those values. Moreover, in most cases, the residual does not follow a monotonous decrease and there are oscillations around the given threshold when the process arrives near the solution. Hence, if we do not take care, we can detect a local convergence too early since, a few iterations later, the residual will go back above the threshold, leading to a false detection of local convergence. This is a common problem of iterative algorithms.

Currently, there is no way to ensure a definitive local convergence on a processor without modifying the iterative process, as in [16]. 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 accuracy threshold. This is the mechanism we use in Algorithm 1. It implies the use of a constant, called *THRESHOLD_LCV*, which represents this required number of successive iterations under the residual threshold to assure local convergence (*LCV*). It is important to note that this *THRESHOLD_LCV* value theoretically exists and is finite since, by hypothesis, the asynchronous iterative process converges.

4.2 Global Convergence Detection

As said before, the idea is to use a scheme which is quite similar to the leader election protocol [21]. The latter consists in designating one processor to perform a given task. In our case, the global detection will be detected on one processor, which will then propagate it to all its neighbors, and so on in all the connection graph.

Our election process works with what we call *partialCV* messages between processors. Such a message tells the receiver that all the processors in the subgraph depending on the sender (behind the sender) have reached local convergence. Hence, on each processor, the algorithm considers the number of neighbors which have not sent their *partialCV* message to it yet. This is represented by the *nbNeigNotLCV* variable in Algorithm 1. Then, when this number becomes equal to one and the processor has reached local convergence, it sends a *partialCV* message to its last neighbor which has not sent him such a message yet. It should be noticed that, as we use a spanning tree of the communication graph to perform the leader election, there always exists one node at least in the system which satisfies *nbNeigNotLCV=1*. Thus, the propagation of convergence detection will start as soon as such a node enters in local convergence.

Finally, the processor which is in local convergence and which has received *partialCV* messages from all its neighbors (*nbNeigNotLCV=0*) detects the global convergence. This processor will then broadcast the global convergence to its neighbors, which will forward it to their neighbors, and so on.

Our decentralized method for global convergence detection is given in Algorithm 1. For clarity, a description of the variables used in this algorithm is given in Table 1.

TABLE 1
Description of Variables Used in Algorithm 1

nbNeigNotLCV	number of neighbors which have not sent a partialCV message to the current processor yet
nbIterPreLocalCV	number of successive iterations performed under the residual threshold
preLocalCV	boolean being true when an iteration is under the residual threshold
localCV	true when local convergence is achieved
globalCV	true when global convergence is achieved

Algorithm 1 Decentralized global convergence detection
for all $P_i, i \in \{1, \ldots, N\}$ **do**
  nbNeigNotLCV $\leftarrow$ nbNeighbors
  nbIterPreLocalCV $\leftarrow$ 0
  preLocalCV $\leftarrow$ false
  localCV $\leftarrow$ false
  globalCV $\leftarrow$ false
  repeat
    if $\neg$ localCV **then**
      ... iterative process and ...
      ... evaluation of preLocalCV ...
      if preLocalCV **then**
        nbIterPreLocalCV $\leftarrow$ nbIterPreLocalCV + 1
        if nbIterPreLocalCV $=$ THRESHOLD_LCV **then**
          localCV $\leftarrow$ true
        end if
      else
        nbIterPreLocalCV $\leftarrow$ 0
      end if
    end if
    if localCV **then**
      if nbNeigNotLCV $=$ 0 **then**
        globalCV $\leftarrow$ true
      else
        if nbNeigNotLCV $=$ 1 **then**
          send a partialCV message to the last neighbor
          corresponding to nbNeigNotLCV
        end if
      end if
    end if
  until globalCV
  BroadCast a globalCV message to all neighbors from
  which no globalCV message arrived
end for

Algorithm 2 function recvPartialCV()
nbNeigNotLCV $\leftarrow$ nbNeigNotLCV - 1

Algorithm 3 function recvGlobalCV()
globalCV $\leftarrow$ true

Receipts of messages have been placed in separated functions and do not directly appear in the main algorithm, since they may happen at any time during an iteration of the main iterative loop. So, they are managed independently of the main process. This is particular to asynchronous iterative algorithms where communications are not performed and managed at specific times in the algorithm but must be treated as soon as they occur.

Function recvPartialCV() only consists in decreasing the number of neighbors which have not reached local convergence yet. Function recvGlobalCV() consists in stopping the iterative process on the node by setting the *globalCV* value to *true*.

5 Proof of the Algorithm

We recall that our detection algorithm is to be used with totally asynchronous algorithms. Within the scope of this paper, we consider any asynchronous iterative process which converges. This is achieved under conditions given in Section 3, which are verified in a large class of scientific computations.

5.1 Preliminary Definitions

Let $P = \{P_1, \ldots, P_N\}$ be the set of the processors.

Let us define $\mathrm{NOpCVmess}(P_i, P_j, t)$ between two neighboring processors P_i and P_j at time t as:

$$\mathrm{NOpCVmess}(P_i, P_j, t) = \begin{cases} \text{true} & \text{if } P_i \text{ has not received a partialCV} \\ & \text{message from } P_j \text{ yet} \\ \text{false} & \text{if } P_i \text{ has received a partialCV message from } P_j. \end{cases}$$

Our 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 (*nbNeigNotLCV*). Since these properties evolve in function of time, 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:

$$nbNeigNotLCV = k$$
$$localCV = \begin{cases} \text{true} & \text{if } e = c \\ \text{false} & \text{if } e = d. \end{cases}$$

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 form P_j to P_k at time t (t is included because communication times may vary during the process).

We have then:

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

5.2 Decentralized Global Convergence Detection Theorem

Theorem 1. *If the following hypotheses are satisfied:*

- *(H1) The communication graph used for the detection 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

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

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

5.3 Proof of the Theorem

We propose to make the proof in two steps:

- (A) We prove that $S_0^c(t_d) \neq \emptyset$ implies all the other statements of Theorem 1.
- (B) We prove that $\exists t_d \in \mathbb{N}$ such that $S_0^c(t_d) \neq \emptyset$.

Part (A). Let us define $\mathrm{Neigh}(P_i)$ the set of physical neighbors of processor P_i. In order to get the processor P_i in $S_0^c(t)$, by Algorithm 1 we must have

$$\forall P_j \in \mathrm{Neigh}(P_i),\ \mathrm{NOpCVmess}(P_i, P_j, t) = false,$$

which implies in turn for all the P_j that

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

and, by recursion, we can deduce that

$$\begin{aligned} \forall P_a \in P(t) \setminus \{P_i\}, \quad & \exists P_b \in P(t), \\ & \mathrm{NOpCVmess}(P_b, P_a, t) = false. \end{aligned} \tag{2}$$

This means that all the P_a in this equation have sent a *partialCV* message to the corresponding P_b and, by Algorithm 1, this is only possible once P_a has reached local convergence.

Thus, we have

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

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

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

Moreover, by Algorithm 1, we also know that the condition for a processor P_a to verify (2) (sending of a *partialCV* message to another node) is to have its *nbNeigNotLCV* equal to one.

Hence, it comes that

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

and, then,

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

and all the other statements of Theorem 1 are verified.

Part (B). By definition, at the beginning of the process, the following statements are verified:

$$\begin{aligned} &S_k^c(0) = \emptyset \quad \forall k \in \{0, \ldots, N-1\}, \\ &S_0^d(0) = \emptyset, \\ &S_1^d(0) \neq \emptyset. \end{aligned} \tag{3}$$

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

$$\begin{aligned} &P_i \in S_k^d(t),\ i \in \{1, \ldots, N\},\ k \in \{0, \ldots, N-1\} \\ &\Rightarrow \exists\, t_c(i) \in \mathbb{N},\ \forall t \geq t_c(i), P_i \in \bigcup_{u=0}^{k} S_u^c(t), \end{aligned} \tag{4}$$

hence,

$$\exists t'(k) \in \mathbb{N}, \forall t \geq t'(k),\ |S_k^d(t)| = 0,\ k \in \{0, \ldots, N-1\}. \tag{5}$$

Equations (3), (5), and Algorithm 1 imply that

$$\exists t_{dn},\ \begin{cases} S_1^c(t_{dn}) \neq \emptyset \\ \forall t < t_{dn},\ \bigcup_{u=0}^{N-1} S_u^d(t) \neq \emptyset \\ \forall t \geq t_{dn},\ \bigcup_{u=0}^{N-1} S_u^d(t) = \emptyset \\ \forall t < t_{dn},\ S_0^c(t) = \emptyset. \end{cases} \tag{6}$$

The last statement is, in fact, a deduction of 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 (6) that $S_1^c(t_{dn}) \neq \emptyset$.

So, every $P_i \in S_1^c(t_{dn})$, according to Algorithm 1, sends a *partialCV* message to its unique neighbor P_k which verifies $\mathrm{NOpCVmess}(P_i, P_k, t_{dn}) = \mathrm{true}$.

If we define

$$\begin{aligned} A(t) = \{&P_i \in S_1^c(t),\ \exists! P_k \in P(t) \text{ such that} \\ &\mathrm{NOpCVmess}(P_i, P_k, t) = \mathrm{NOpCVmess}(P_k, P_i, t) = \mathrm{true}\} \end{aligned}$$

and

$$\begin{aligned} B(t) = \{&P_k \in P(t),\ \exists P_i \in A(t) \text{ such that} \\ &\mathrm{NOpCVmess}(P_i, P_k, t) = \mathrm{true}\}. \end{aligned}$$

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 arrived at time t yet.

From (H1), we deduce the following Lemma.

Lemma 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 1:

Since $t' \geq t_{dn}$, we are in the context of (6) where all the processors are in the subsets $S_u^c, u \in \{0, \ldots, N-1\}$.

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 received the *partialCV* message from at least two of its neighbors yet.

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 1: One consequence is that as soon as the set A becomes empty, it cannot become nonempty again.

Remark 2: Another consequence is that time t' is equivalent to time t_d in Theorem 1 since $S_0^c(t') \neq \emptyset$ and then Part (A) of the proof implies all the other statements of the theorem.

Remark 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 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 1 as pointed out by Remark 2.

If it is nonempty, (6) 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 $B(t_{dn})$:

$$\begin{aligned} &(1)\ \forall P_l \in B(t_{dn}), P_l \in S_1^c(t_{dn}), \\ &(2)\ \forall P_l \in B(t_{dn}), P_l \in \bigcup_{u=2}^{N-1} S_u^c(t_{dn}). \end{aligned} \tag{7}$$

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 $\mathrm{NOpCVmess}(P_l, P_i, t_{dn}) = \mathrm{true}$ and $\mathrm{NOpCVmess}(P_l, P_i, t_r(l,i)) = \mathrm{false}$ 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 $S_u^c(t_{dn})$, $u \in \{2, \ldots, N-1\}$ such that $P_l \in S_u^c(t_{dn})$, and then by Algorithm 1,

$$\begin{aligned} &P_l \in \bigcup_{v=0}^{u-1} S_v^c(t_r(l,i)),\ \text{with } i \text{ such that} \\ &P_i \in A(t_{dn}) \text{ and } \mathrm{NOpCVmess}\,(P_i, P_l, t_{dn}) = \mathrm{true}. \end{aligned} \tag{8}$$

This means that each time a processor receives a *partialCV* message, its number of neighbors which have not sent him a *partialCV* message yet decreases by one. Moreover, we use a union of the first $u-1$ 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 (7) 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 (8) ensures that this recursion will empty all the subsets $S_u^c(t), u \in \{2,\ldots,N-1\}$ 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 1 by Remark (2).

As a last remark, we can note 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 1).

6 Practical Version of the Algorithm

The major problem of Algorithm 1 comes from the knowledge of the minimal number of iterations necessary to ensure local convergence on each node. In real cases, it is not possible to evaluate it and the value of *THRESHOLD_LCV* cannot be precisely known.

This constant is the critical point of our algorithm since the local convergence detection directly depends on it and, consequently, the global detection too. As a matter of fact, if it is set smaller than the actual number of iterations necessary to ensure local convergence, the algorithm may detect false local convergences leading to a false global convergence detection. On the other hand, if it is set larger than the correct value, local convergences will be correctly detected as well as the global one, but with extra delays which will decrease the efficiency of the algorithm. Moreover, the value of this constant also depends on the desired final accuracy since it is obviously easier, and then faster, to reach low accuracies than high ones.

Thus, even if Algorithm 1 is valid and theoretically allows the detection of global convergence, it is not practically usable under this form since, for a given value of *THRESHOLD_LCV*, it may work in some particular cases, but not for all of them. In order to overcome this problem, we are brought to consider our practical algorithm in the partially asynchronous context where the delays are bounded by a given integer B such that:

$$\exists B \in \mathbb{N}, \text{ such that } t - B < r_j^i(t) \leq t. \tag{9}$$

Contrary to the appearances, this is not a strong weakness of our algorithm since, in practice, delays are always finite. For arbitrarily large delays, the completion of the iterative process is not ensured in a finite time which does not present a great practical interest.

According to the first algorithm, we then introduce additional mechanisms around the local convergence detection:

Our practical algorithm, presented in Algorithm 4, uses almost the same method to detect local convergence as in Algorithm 1. Nonetheless, since, in practice, we cannot be sure that local convergence is actually reached, it does not stop the iterative process when an eventual local convergence is detected but continues to compute the iterations and to update *preLocalCV*.

In this new version, the *THRESHOLD_LCV* value is replaced by a *THRESHOLD_SLCV* value, which stands for *supposed* local convergence (*SLCV*) and represents the number of iterations after which we suppose that local convergence has been reached. This value can then be set arbitrarily. *partialCV* messages are replaced by *sPartialCV* messages (the *s* is for *supposed*) whose sendings are based upon these supposed local convergences.

With this mechanism, it is possible to have *preLocalCV* which becomes true and which, a few iterations later, becomes false again if *THRESHOLD_SLCV* is smaller than the actual threshold. So, when this happens, the supposed local convergence is canceled on the node and a new kind of message (a *cancelSPartialCV* message) is sent to the same destination processor as the previous *sPartialCV* message to alert that local convergence is not achieved anymore in this part of the graph.

Everyone may have noted that if *THRESHOLD_SLCV* is greater than or equal to the theoretical threshold, we come back to the context of Algorithm 1 and there will not be any cancellation message during the execution of the algorithm and the global detection is correctly detected. Nevertheless, in practical cases, the *THRESHOLD_SLCV* will not be chosen too large, which is unlikely to correspond to the theoretical value.

Finally, when a processor receives a cancellation message, it increases its count of neighbors which are not in local convergence yet which will prevent its own sending of a *partialCV* message (if not performed yet). Nevertheless, it is possible that the cancellation message arrives too late on a node and that the *partialCV* message has already been sent by this node. In that case, the cancellation message is forwarded to the destination node of the *partialCV* message, and so on, in order to cancel all the diffusion of the false detection.

Algorithm 4 Practical version of Algorithm 1

```
for all P_i, i ∈ {1,...,N} do
  nbNeigNotLCV ← nbNeighbors
  nbIterPreLocalCV ← 0
  preLocalCV ← false
  sLocalCV ← false
  globalCV ← false
  repeat
    ... iterative process and ...
    ... evaluation of preLocalCV ...
    if ¬ sLocalCV then
      if preLocalCV then
        nbIterPreLocalCV ← nbIterPreLocalCV + 1
        if nbIterPreLocalCV = THRESHOLD_SLCV then
          sLocalCV ← true
        end if
      else
        nbIterPreLocalCV ← 0
      end if
    else
      if ¬ preLocalCV then
        sLocalCV ← false
        nbIterPreLocalCV ← 0
        if an sPartialCV message has already been sent then
```

```
        send a cancelSPartialCV message with the current
        iteration number to the same destination neighbor
      end if
    else
      if nbNeigNotLCV = 0 then
        globalCV ← true
      else
        if nbNeigNotLCV = 1 then
          send an sPartialCV message with the current
          iteration to the last neighbor corresponding to
          nbNeigNotLCV
        end if
      end if
    end if
  end if
  until time period with globalCV > MaxTraversalTime
  BroadCast a globalCV message to all neighbors from
  which no globalCV message arrived
end for
```

Unfortunately, this is not sufficient yet to avoid all the false global convergence detections. For example, let's consider that all the nodes but two have reached their local convergence. For clarity, let's call them A and B. When one of the two remaining nodes goes into the *sLocalCV* state (let's say A), it then sends its *sPartialCV* message. But, if a few iterations later it cancels its *sLocalCV* state and sends a *cancelSPartialCV* message, the false global convergence detection will be avoided on B only if the cancellation message arrives on B *before* it has left its iterative loop.

Obviously, nothing ensures us that this message will arrive soon enough to avoid the false detection.

Nevertheless, if we can evaluate the maximal communication time on each link in the communication graph and if these times are finite, it is then possible to avoid the false detections by adding an additional condition for leaving the iterative loop.

This condition consists in waiting for a particular amount of time after the supposed global convergence has been found on the elected node. This waiting is expressed in the condition of the main iterative loop of Algorithm 4. This amount of time (represented by *MaxTraversalTime* in the algorithm) is the maximal time needed for a message to go from one node to another one in the communication graph. Since, in this context, all the maximal communication times between neighbors are known and finite, this *traversal* time is also finite and can be computed.

Hence, by this scheme, we take into account all cancellation messages sent before the time at which global convergence is normally detected on the elected node, including those which are in transit. When no cancellation message arrives on the elected node during this additional time interval, global convergence can be surely assumed.

It is interesting to see that the knowledge of maximal time communication is not only required in the decentralized algorithm, but also in the centralized one (between the master and all other nodes) to ensure the coherence of the detection. However, in practical cases, an upper bound of this maximal traversal time can always be evaluated.

In fact, with this mechanism, we ensure that the global convergence we detect actually corresponds to a time at which *all* the nodes of the system are in local convergence. Finally, this is exactly the same halting criterion as in sequential iterative algorithms with the same assumptions upon local convergence. The use of a threshold to detect the local convergences can also be seen as a moderation of the number of messages sent for the global convergence detection.

TABLE 2
Arrays Used for Order Verifications in Functions recvSPartialCV() and recvCancelSPartialCV()

prevIterNumS[N]	array indicating, for each neighbor, the greatest iteration number received in sPartialCV messages until current time. Initialized to -1.
prevIterNumC[N]	array indicating, for each neighbor, the greatest iteration number received in cancelSPartialCV messages until current time. Initialized to 0.

As for Algorithm 1, receipts have been placed in separated functions and do not directly appear in the main algorithm. But, a new problem, induced by having two kinds of messages concerning the local convergences, arises with the order of these receipts. In some communication systems, the order of the messages may not be respected, especially in the context of asynchronous algorithms. Thus, the modifications of the variables *nbNeigNotLCV* and *globalCV* in the receipt functions given in Algorithm 5 and Algorithm 6 must be performed only when the received message actually corresponds to a more recent state on the source node.

A simple solution to this problem is to associate to each message the number of the iteration at which it is sent from its source node, in order to be able to reorder the messages on the destination node. So, for each receipt of *sPartialCV* and *cancelSPartialCV* messages, the modifications of *nbNeigNotLCV* and *globalCV* will only occur when the current message actually represents a more recent change of state on the source node. These checkings correspond to the first test in both Algorithms 5 and 6. The second test only consists in updating the previous iteration number from this source node, if necessary. This mechanism implies the use of additional arrays described in Table 2. For each kind of message, they contain the greatest (and then the most recent) iteration numbers received from each neighbor. Their size is set to N which is not optimal, but allows to directly map the ranks of the processors on the array indices.

Algorithm 5 function recvSPartialCV()

```
srcNode ← number of the source node of the message
currentIterNum ← iteration number of source node
if prevIterNumS[srcNode] < prevIterNumC[srcNode]
and prevIterNumC[srcNode] < currentIterNum then
  nbNeigNotLCV ← nbNeigNotLCV - 1
end if
if prevIterNumS[srcNode] < currentIterNum then
  prevIterNumS[srcNode] ← currentIterNum
end if
```

Algorithm 6 function recvCancelSPartialCV()

```
srcNode ← number of the source node of the message
currentIterNum ← iteration number of source node
if prevIterNumC[srcNode] < prevIterNumS[srcNode]
and prevIterNumS[srcNode] < currentIterNum then
  nbNeigNotLCV ← nbNeigNotLCV + 1
```

```
    globalCV ← false
  end if
  if prevIterNumC[srcNode] < currentIterNum then
    prevIterNumC[srcNode] ← currentIterNum
  end if
```

Concerning the function recvGlobalCV(), it is not shown here since it is exactly the same as in Algorithm 3.

In short, the whole mechanism described by Algorithm 4 and its additional functions prevent false global detections by canceling *partialCV* messages when it finds false local convergence detections. Moreover, it does not add an important overhead after the detection of the right global convergence.

7 Experiments

The major interest of such an algorithm is to dramatically enhance the flexibility of deployment of parallel iterative algorithms. Effectively, when performing grid computing in a global context, a very common case is to use several distant clusters. Nevertheless, due to access constraints (firewalls, private networks, etc.), all the machines in a given cluster may not be able to communicate with any other machine in another cluster. Hence, a classical centralized convergence detection where each node sends state messages to a master node cannot be used.

The centralized detection can, however, still be used if explicit forwarding of state messages is done between neighboring nodes of the system. That is to say, when a node needs to send a message to the master, it sends the message to one of its neighbors which is closer to the master and this neighbor will forward the message to one of its neighbors yet more closer to the master than itself, and so on, until reaching the master. This version has the advantage to require no modification of the detection process on the master node. Nonetheless, it is clearly more expensive in terms of communications and delay to detect the global converge.

Finally, it is interesting to compare the efficiency of the decentralized version with the two other versions, the centralized one and the forwarding one, in two classical contexts of grid computing (local and distant). These experiments will allow us to deduce what kind of detection algorithm to use in a given computing context.

By efficiency, we mean the delay required to detect the global convergence. There is no direct way to measure it since the time at which the global detection actually begins cannot be evaluated. Hence, in order to estimate the relative efficiencies of the detection algorithms, we compare the execution times of three variants of the same parallel iterative algorithm. Each of them using a different convergence detection algorithm.

To be able to test the centralized version, the grid contexts used have no access constraints (due to security, for example). An application typically solved by iterative algorithms has been chosen for our tests. It is a nonlinear chemical problem called the advection-diffusion problem.

7.1 Description of the Nonlinear Problem

In this problem, we want to compute the evolutions of the concentrations of two chemical species in a two-dimensional domain. It is solved by using a discretization of the space on a two-dimensional grid: x and z. The evolutions are given for each point (x, z) of the grid for a given time interval by differential equations:

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial z} K_v(z) \frac{\partial c^i}{\partial z} + R^i(c^1, c^2, t), \quad (10)$$

where $i = 1, 2$ denotes the number of the chemical species and

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2, \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 + q_4(t) c^2, \end{aligned} \quad (11)$$

with

$$\begin{array}{lcll} K_h & = & 4.0 \times 10^{-6} & V = 10^{-3} \\ K_v(z) & = & 10^{-8} e^{\frac{z}{5}} & c^3 = 3.7 \times 10^{16} \\ q_1 & = & 1.63 \times 10^{-16} & q_2 = 4.66 \times 10^{-16} \\ q_j(t) & = & e^{-a_j / sin(\omega t)} & \text{for } sin(\omega t) > 0 \\ q_j(t) & = & 0 & \text{otherwise} \end{array}$$

and $j = 3, 4$, $\omega = \pi/43,200$, $a_3 = 22.62$, and $a_4 = 7.601$.

The time interval is $[0s, 7,200s]$ and the initial conditions are the following:

$$\begin{aligned} c^1(x, z, 0) &= 10^6 \alpha(x) \beta(z), \\ c^2(x, z, 0) &= 10^{12} \alpha(x) \beta(z), \end{aligned} \quad (12)$$

with

$$\begin{aligned} \alpha(x) &= 1 - (0.1x - 1)^2 + (0.1x - 1)^4/2, \\ \beta(z) &= 1 - (0.1z - 1)^2 + (0.1z - 4)^4/2. \end{aligned} \quad (13)$$

The discretization along x and z allows us to rewrite the system of PDEs (Partially Differential Equations) in a system of ODEs (Ordinary Differential Equations):

$$\frac{dy(t)}{dt} = f(y(t), t) \quad \text{with} \quad y = (c^1, c^2). \quad (14)$$

The global solution of (14) is computed using the finite differences scheme. A main loop is made over the time steps, the Euler implicit method is used to discretize the system along each time interval and the iterative Newton method is used to linearize the obtained nonlinear system. There are two main strategies to perform the Newton method. The first one consists in applying the Newton method on the entire system and using an asynchronous parallel linear solver over the global system. Unfortunately, several synchronizations are necessary at each time step. In the second approach, called multisplitting Newton, the system is decomposed into several subsystems which are locally solved by a sequential linear system solver according to data dependencies between the subsystems. In this method, only one synchronization is needed at each time step (see [22] for further information about this method). In our case, the multisplitting approach has been chosen with the GMRES method [1] as the sequential linear solver.

An AIAC algorithm has been designed to solve this problem and three variants have been written according to the convergence detection algorithm used. In each program, only the convergence detection is different.

The particular interest of this application in our evaluation is that it uses a series of iterative processes separated by synchronizations. In each iterative process, the convergence detection algorithm is used and then, the impact of its efficiency over the overall execution time is more important

TABLE 3
Total Execution Times (Seconds) of the AIAC Algorithm with the Three Convergence Detection Algorithms

Grid context	Centralized	Forwarded	Decentralized
Distant sites (400)	163	162	155
Distant sites (600)	649	651	639
Local site (800)	368	373	373

than in a single iterative process. So, this is a convenient example to point out eventual differences among the convergence detection algorithms.

7.2 Results

Two grid contexts have been used to perform our experiments. The first one is composed of 10 machines: one pentium III 833Mhz, one pentium III 900Mhz, one Pentium IV 1.7Ghz, three Athlon XP 2000, two Pentium IV 2.4Ghz, one Pentium IV 2.6Ghz, and one Athlon XP 2800, scattered over four distant sites linked with 10Mbits/s Ethernet and ADSL 128Kb/s. The second one is a local cluster of forty PCs linked with 100Mbits/s Ethernet containing three kinds of processor: 15 Duron 800Mhz, 10 Pentium IV 1.7Ghz, and 15 Pentium IV 2.4Ghz.

Our experiments have been done with no other users on the machines. Moreover, different sizes of the nonlinear chemical problem have been used on each grid context to take into account the different numbers of machines and then to have a sufficient amount of computation. Discretization grids of 400×400 and 600×600 points have been used for the distant cluster and a larger grid of 800×800 points has been used for the local cluster which contains more machines. Using different grid sizes is not a problem since the focus point of those experiments is the difference between execution times of the three tested algorithms and not their absolute values.

All our algorithms have been implemented using the free Corba [23] ORB OmniOrb 4. It is a robust and high performance ORB which is certified to be in compliance with Corba 2.1. The most interesting features of this programming environment for the implementation of our algorithms are the thread support and the remote procedure call (RPC) mechanism which allow us to efficiently perform the asynchronous communications. Moreover, this choice is also related to the facility of deployment of this environment compared to more classical communication libraries like MPI.

The results (averages of a series of ten executions) obtained for the three variants of convergence detection are given in Table 3. They represent the total execution times (computation and convergence detection) of the algorithm.

A first question which may arise is: What about the interest of using machines with different powers? In other words: Would the execution be faster on the fastest machine alone? In fact, it is not the case since although the slower machines tend to slow down the global parallel progress (compared to a cluster with the same number of machines but only containing the fastest model), they contribute to process multiple data at the same time and then to be faster than one machine alone. For example, the execution on the fastest machine with the grid size 600×600 takes about 1,987 seconds. Moreover, there is also the advantage of increasing the amount of resources to solve larger problems. For example, the execution of the algorithm with the grid size 800×800 is not possible on a single machine due to memory lack.

In the local context, the centralized version is better since communications to the master are very fast. On the opposite, in the distant context, it is the decentralized version which always obtains the best performances. This comes from the fact that this method only uses communications between neighbors and does not force the detection over a given node, which is the case in the two other versions.

However, the interesting result is that the forwarding version is substantially less efficient than the decentralized one in the distant context. This is very interesting since in such cases, only those two versions can be used due to access constraints.

Moreover, even better results could be expected for the decentralized version with more distant machines. This is essential for future trends of large scale grid computing.

Thus, the decentralized algorithm is not only interesting for the deployment of parallel iterative algorithms, but can also bring better performances in the context of grid computing by reducing the latency of global convergence detection.

8 Conclusion

Theoretical and practical versions of a decentralized algorithm for global convergence detection in parallel iterative algorithms have been presented.

The theoretical version of this algorithm has been proven to work correctly on all totally asynchronous converging iterative algorithms under two classical assumptions in this domain: The communication graph used is connected and acyclic, and the communications between processors are performed in a finite time.

The proposed algorithm is very useful in the context of grid computing where the processors are distributed and where detecting the convergence on a master processor may be penalizing or even impossible. This is the case for large clusters with geographically distributed machines. It improves the performances of iterative algorithms especially if the communication/computation ratio is not negligible.

It has been discussed why the theoretical algorithm cannot be directly used in practice for totally asynchronous algorithms. Nevertheless, the practical version proposed fully works on partially asynchronous algorithms and may eventually work on particular cases of totally asynchronous ones. All the modifications from the theoretical version to the practical version have been detailed and justified. It has also been shown that the practical version uses an ending mechanism similar to the one used for sequential iterative algorithms.

This algorithm presents several advantages. The first one is to avoid communication bottlenecks which are inherent to centralized versions. The second one is a larger flexibility in the context of global grid computing since it does not require a strongly connected graph, but only an acyclic graph. Such a graph can always be extracted from the dependency graph of the system. This is particularly convenient for deployments through firewalls where some

of the processors used in the computation may be accessible only through one machine. Experimental results have pointed out that, in such distant contexts, the decentralized algorithm reduces the latency of the convergence detection and then improves the overall performances of parallel iterative algorithms. Moreover, our method does not involve any modification of the iterative process as opposed to [8]. This has the double advantage to be easier to implement and also to work surely when the iterative process converges, which is not always the case with methods modifying the initial iterative process. Finally, since we do not merge messages for the computation and messages for the convergence detection, our method is tolerant to the loss of computational messages.

Finally, this decentralized algorithm for global convergence detection is an important step toward the full adaptation of parallel iterative algorithms in global grid computing, especially for AIACs. The power of these last algorithms has already been shown in several previous studies such as [2] and [17] and, therefore, the next step is to study a powerful and convenient environment for the deployment of an AIAC algorithm on a global cluster.

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems,* second ed. SIAM, 2003.

[2] J.M. Bahi, S. Contassot-Vivier, and R. Couturier, "Asynchronism for Iterative Algorithms in a Global Computing Environment," *Proc. 16th Ann. Int'l Symp. High Performance Computing Systems and Applications (HPCS 2002),* pp. 90-97, June 2002.

[3] N. Francez, "Distributed Termination," *ACM Trans. Programming Languages and Systems,* vol. 2, no. 1, pp. 42-55, Jan. 1980.

[4] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. vanGasteren, "Derivation of a Termination Detection Algorithm for Distributed Computation," *Information Processing Letters,* vol. 16, no. 5, pp. 217-219, 1983.

[5] S.P. Rana, "A Distributed Solution to the Distributed Termination Problem," *Information Processing Letters,* vol. 17, pp. 43-46, July 1983.

[6] N. Maillard, E.M. Daoudi, P. Manneback, and J.-L. Roch, "Contrôle Amorti des Synchronisations pour le Test d'Arrêt des Méthodes Itératives," *Renpar 14,* pp. 177-182, Apr. 2002.

[7] D.P. Bertsekas and J.N. Tsitsiklis, "Convergence Rate and Termination of Asynchronous Iterative Algorithms," *Proc. 1989 Int'l Conf. Supercomputing,* pp. 461-470, June 1989.

[8] S.A. Savari and D.P. Bertsekas, "Finite Termination of Asynchronous Iterative Algorithms," *Parallel Computing,* vol. 22, pp. 39-56, 1996.

[9] D. El Baz, *Contribution à l'Algorithmique Parallèle, le Concept d'Asynchronisme: Étude Théorique, Mise en Oeuvre, et Application.* LAAS CNRS, Institut National Polytechnique de Toulouse, 1998.

[10] A.S. Charão, "Multiprogrammation Parallèle Générique des Méthodes de Décomposition de Domaine," PhD dissertation, Institut National Polytechnique de Grenoble, http://www-mediatheque.imag.fr/Mediatheque.IMAG/collections-electroniques/publications/index.html, 2001.

[11] "IEEE Standard for a High Performance Serial Bus,"Technical Report Std 1394-1995, IEEE Computer Society, Aug. 1996.

[12] G. Antonoiu and P.K. Srimani, "A Self-Stabilizing Leader Election Algorithm for Tree Graphs," *J. Parallel and Distributed Computing,* vol. 34, no. 2, pp. 227-232, May 1996.

[13] M. El-Ruby, J. Kenevan, R. Carison, and K. Khalil, "Leader Election in Distributed Computing Systems," *Proc. Conf. Computing in the 90's,* pp. 350-356, Oct. 1991.

[14] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Trans. Computers,* vol. 31, no. 1, pp. 47-59, Jan. 1982.

[15] S.D. Stoller, "Leader Election in Asynchronous Distributed Systems," *IEEE Trans. Computers,* vol. 49, no. 3, pp. 283-284, citeseer.ist.psu.edu/5897.html, Mar. 2000.

[16] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods.* Englewood Cliffs, N.J.: Prentice Hall, 1989.

[17] J.M. Bahi, S. Contassot-Vivier, and R. Couturier, "Coupling Dynamic Load Balancing with Asynchronism in Iterative Algorithms on the Computational Grid," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS 2003),* p. 40, Apr. 2003.

[18] M. El Tarazi, "Some Convergence Results for Asynchronous Algorithms," *Numerical Math.,* vol. 39, pp. 325-340, 1982.

[19] D. El Baz, P. Spiteri, J.C. Miellou, and D. Gazen, "Asynchronous Iterative Algorithms with Flexible Communication for Nonlinear Network Flow Problems," *J. Parallel and Distributed Computing,* vol. 38, no. 1, pp. 1-15, Oct. 1996.

[20] J.M. Bahi, "Asynchronous Iterative Algorithms for Nonexpansive Linear Systems," *J. Parallel and Distributed Computing,* vol. 60, no. 1, pp. 92-112, Jan. 2000.

[21] N. Lynch, *Distributed Algorithms.* San Francisco, Calif.: Morgan Kaufmann, http://theory.lcs.mit.edu/tds/distalgs.html, 1996.

[22] J.M. Bahi, J.-C. Miellou, and K. Rhofir, "Asynchronous Multisplitting Methods for Nonlinear Fixed Point Problems," *Numerical Algorithms,* vol. 15, nos. 3 and 4, pp. 315-345, 1997.

[23] A. Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture.* Reading, Mass.: Addison-Wesley, Dec. 1997.

Jacques M. Bahi received the PhD degree and an HDR in applied mathematics from the University of Franche-Comté (France). He is a full professor of computer science at the same university. His research interests include asynchronism, distributed computing, load balancing and massive parallelism. He is a member of the IEEE and the IEEE Computer Society.

Sylvain Contassot-Vivier received the PhD degree from École Normale Supérieure de Lyon in 1998. He joined the Image Processing Group of the ERIC Laboratory at the University Lyon 2 in 1997. He obtained a postdoctoral position at the FUNDP of Namur (Belgium) in 1998. Since 1999, he has been an assistant professor in the Computer Science Laboratory LIFC at the University of Franche-Comté (France). His main research interests lie in the parallel computing domain such as massively parallel systems, asynchronism, grid-computing, and also in the domain of image processing and vision. He is a member of the IEEE and the IEEE Computer Society.

Raphaël Couturier received the PhD degree in computer science from Henri Poincaré University, Nancy, France, in January 2000. In September 2000, he joined the Computer Science Laboratory of the University of Franche-Comté, where he is an assistant professor. His research interests include parallel and distributed computation, numerical algorithms, and data mining. He is a member of the IEEE and the IEEE Computer Society.

Flavien Vernier graduated from the LIFC of University of Franche-Comté (France) in 2001. Since 2001, he has been a PhD student at the same university. His main scientific interests lie in the parallel computing domain such as distributed load balancing, distributed numerical algorithms, and asynchronism.