An Efficient and Robust Decentralized Algorithm for Detecting the Global Convergence in Asynchronous Iterative Algorithms

Jacques M. Bahi[1], Sylvain Contassot-Vivier[2], and Raphaël Couturier[1]

[1] LIFC, University of Franche-Comte, Belfort, France
{jacques.bahi,raphael.couturier}@univ-fcomte.fr,
http://info.iut-bm.univ-fcomte.fr/
[2] LORIA, University Henri Poincaré, Nancy, France
sylvain.contassotvivier@loria.fr
http://www.loria.fr/~contasss/homeE.html

Abstract. In this paper we present a practical, efficient and robust algorithm for detecting the global convergence in any asynchronous iterative process. A proven theoretical version, together with a first practical version, was presented in [1]. However, the main drawback of that first practical version was to require the determination of the maximal communication time between any couple of nodes in the system during the entire iterative process. The version presented in this paper does not require any additional information on the parallel system while always ensuring correct detections.

Keyword: Asynchronous iterative algorithms, convergence detection.

1 Introduction

Iterative algorithms are very well suited to numerous problems in the context of scientific computations. They are often opposed to direct algorithms which give the exact solution of a problem within a finite number of operations whereas iterative algorithms provide successive approximations of it. It is said that they converge (asymptotically) towards the solution. When dealing with very large size problems, iterative algorithms are preferred, especially if they give a good approximation in a little number of iterations [2]. In other cases, they represent the only way to solve the problem as, for example, in the polynomial roots finding problem.

For all those reasons, parallel iterative algorithms are very popular. Nevertheless, most of them are synchronous and we showed in [3] that using asynchronism in such parallel iterative algorithms was far more efficient in the emerging contexts of parallelism such as grid computing. In the scope of this study, the term asynchronism means that each processor performs its local computations without waiting for the last updates of data dependencies coming from other processors. The asynchronous algorithms proposed in our first works used a centralized method to detect the global convergence, which was not best suited to grid contexts. In [1], both a theoretical and a practical version of a

J.M.L.M. Palma et al. (Eds.): VECPAR 2008, LNCS 5336, pp. 240–254, 2008.

decentralized global convergence detection algorithm were proposed. The validity of the theoretical version designed for totally asynchronous algorithms was proven, and a practical version working on partially asynchronous algorithms, i.e. asynchronous algorithms with bounded delays, was deduced. However, that last algorithm presented the drawback of requiring additional information on the parallel system which is not easy, if not to say impossible, to collect in practice.

In this paper, we present another practical version of the decentralized algorithm for detecting global convergence that no longer requires any additional information on the system but only the local states of the nodes.

As in the previous version, messages for the computation and messages for the convergence detection are distinguished. This provides some degree of loss tolerance on the computational messages. The messages involved in the convergence detection are quite small and their limited number avoids any representative network overload which could penalize the progress of the iterative process. Finally, the delay of detection after the actual convergence stays reduced compared to the global process.

The following section briefly presents the previous studies related to convergence detection algorithms. Then, in order to be self-contained, the principles of asynchronous iterative algorithms are given in Sect. 3. Section 4 describes the main problems related to the convergence detection and the weaknesses of our previous algorithm. The new practical version of the decentralized algorithm for convergence detection is described in Sect. 5 and evaluated in Sect. 6.

2 Related Works

Most of the previous studies on the convergence detection problem in parallel iterative algorithms (see for example [4]) are based on centralized and/or synchronous algorithms (typically a global reduction), which are neither suited to large scale and/or distant distributed computations nor to the decentralized nature of asynchronous iterative algorithms.

Concerning the specific studies related to asynchronous iterative algorithms, distributed convergence detection was firstly introduced in [5] 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 [6] under rather restrictive hypotheses such as FIFO communications and with modifications of the iterative process itself in order to make it terminate in finite time. Other authors have studied implementations of asynchronous algorithms but always with centralized convergence detection [7].

As in [1], the algorithm presented in this paper is based on a leader election algorithm to manage the termination of asynchronous iterative algorithms in a decentralized way. However, contrary to the practical version presented in that previous paper, the presented practical algorithm does not require any other information apart from the local convergence states of the nodes.

For more information on the distinction between the theoretical and the practical versions of our convergence detection algorithm, and on the leader election protocol, the reader should refer to [1] and the references therein.

3 Asynchronous Iterative Algorithms

Iterative algorithms have the structure $x^{k+1} = g(x^k)$, $k = 0, 1, ...$, with x^0 given, 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.

The parallel version of the iterative algorithm presented above is obtained by the classical block-decomposition of x into m block-components X_i, $i \in \{1, ..., m\}$, and g into a compatible way of m block-components G_i, to reformulate the iterative process as: $X_i^{k+1} = G_i\left(X_1^k, ..., X_m^k\right), i = 1, ..., m$, with X^0 given.

3.1 AIAC Algorithms

AIAC algorithms, which have been introduced in [8], are a variant of the totally asynchronous algorithms. The reader should refer to [9] and [10] to get the two major formulations of the theoretical model of totally asynchronous iterative algorithms. In this paper, we only remind the reader that those algorithms mainly induce the notion of delays between the components of the system. In totally asynchronous iterations, some classical conditions are assumed over those delays in order to ensure that the process actually evolves (see again [9]).

The acronym AIAC stands for Asynchronous Iterations - Asynchronous Communications. It means that 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. Those algorithms give very good results in the global context of grid computing as has been shown in [3]. Nevertheless, a centralized algorithm for detecting global convergence is not well suited to the context of grid computing in which all the nodes may not be directly accessible to each other for security reasons. Moreover, another reason for designing a decentralized convergence detection algorithm is that the most general class of parallel iterative algorithms corresponds to the asynchronous iterative algorithms, which are not centralized by nature.

4 Practical Difficulties with Our Previous Algorithm

The ideal way to detect the global convergence of an asynchronous iterative algorithm is to monitor the evolution of the global state of the system between two consecutive periods. In the field 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 data at least as recent as the beginning of that period. The evolution of the system is measured by the residual which is the distance between the two global states according to an adequately chosen norm. Hence, for any converging process, it has been shown that the residual using the adequate norm monotonously decreases from a period to the following one.

So, the convergence detection should only consist in verifying that this residual becomes small enough (under a convergence threshold).

However, it is quite difficult and penalizing in practice to identify periods at the global level of dynamical systems implemented on distributed systems. So, the method commonly used to detect the global convergence is based on a local notion of convergence. That local convergence is detected according to the local residual between two consecutive iterates of the local block-component according to the chosen norm. The local convergence is assumed when that residual becomes smaller than a given threshold. And the global convergence detection consists in verifying that all the nodes are in local convergence and that those local states have been reached with relevant updates of the respective dependent data. The major problem with this is that the local convergence is quite an artificial notion which is not directly linked to the global convergence. In particular, as the local residual is not computed on the global state-vector, it is subject to slightly less restrictive constraints and it may not monotonously decrease. So, additional mechanisms are required to avoid false detections.

In [1], each processor counts a given number of consecutive iterations for which its residual is under the convergence threshold and, only then, it passes in the local convergence state. Although in theory that number of consecutive iterations, which ensures the local convergence of the node, exists and is finite, it is quite difficult to evaluate in practice. Using an approximate value is possible and greatly reduces the potential false detections. However, it requires a global detection mechanism that takes into account possible divergences of the nodes.

In addition to that problem, which is common to every kind of iterative processes, the communication delays induced by the asynchronism change the behavior of the system and make it even more difficult to perform a correct convergence detection. Typically, a processor may be under the threshold thanks to old versions of data coming from some neighbor processors, which is not representative of its potential stabilization.

In Fig. 1 we illustrate such a false detection. We assume that a processor i has dependencies with processors $i-1$ and $i+1$. We also assume that there is a mechanism that detects the global convergence after a given span of time during which all processors are in the local convergence state. The convergence detection problem lies in the fact that processor 1 does not receive any message from its neighbor (processor 2) during several consecutive iterations. Consequently, its state reaches the local convergence that consequently enables the detection of a global convergence. Nevertheless, processor 1 only uses old messages from processor 2 to enter the local convergence state.

In [1], the global convergence detection mechanism is based on the leader election algorithm in order to obtain a decentralized algorithm. In this way, the local convergences of the nodes are propagated through a spanning tree of the system until they meet on a single node. Moreover, some canceling messages are used either to stop the propagation or to inform the elected node that one node is no longer in local convergence (due to the problem of delays exhibited above). This implies a waiting period after the global convergence detection

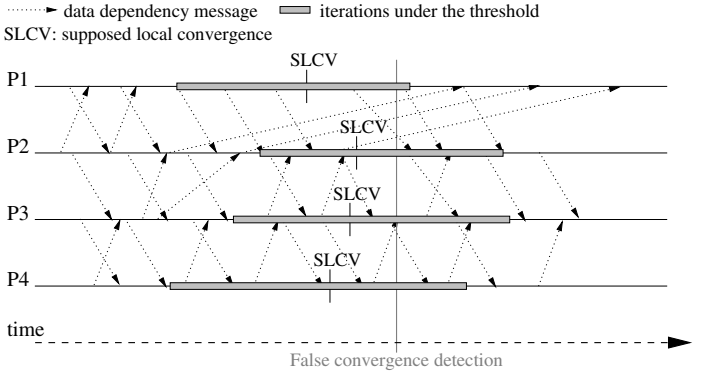

Fig. 1. Example of false convergence detection due to the fact that processor 1 does not receive messages from processor 2 during several consecutive iterations

on the elected node, in order to wait for any potential cancellation message generated in the meantime.

Although that overall detection algorithm works quite well in practice, a criticism can be formulated against it: two constants that depend on the system must be evaluated. The first one is the number of successive iterations under the threshold to assume the local convergence. It indirectly depends on the maximal delays between a processor and its dependencies, and consequently, on the computations performed in the iterative process. The second constant is the maximal traversal time of the system by a cancellation message. It obviously depends on the network configuration of the system. Such information is not easily evaluated accurately in practice.

To bypass those problems, we have designed a new decentralized detection algorithm that does not require any additional information about the system.

5 New Practical Version of the Decentralized Algorithm for Convergence Detection

As seen above, our major problem in the context of asynchronous algorithms is to get a correct image of the global state of the system. Indeed, the possible variations of the local states of the nodes require a robust 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 [1]. Our new version does not require any specific information on the parallel system used. Our approach is closer to the theoretical version presented in our previous work 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, an additional verification phase takes place 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.

The first of the two mechanisms mentioned above 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. The pseudo-period is quite a local version of the periods. For each node, a pseudo-period corresponds to the minimal span of time during which the node receives at least one newer data message from all its dependencies and updates itself. In this way, the local evolution of the node is fully representative between two consecutive pseudo-periods. Thus, the local convergence is assumed only after at least one (but possibly several successive ones) pseudo-period is performed while the residual is under the threshold. This has a far better regulating effect on the local convergence detections in practice and, if it cannot avoid all the false local detections, it strongly limits them.

The second mechanism takes place at the global level of the system, when the global convergence is detected. As in our previous algorithm, the global detection is performed by a leader-election-like algorithm, according to the local convergences of the nodes. However, instead of using cancellation messages when the state of a node changes, as in our previous version, our new process lets the global detection occur. Nonetheless, a new step is added after that global detection which consists in verifying that all the nodes were actually in local convergence at the time of the detection and that their states were representative of their evolutions. That additional step is decomposed into four 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. The actual global convergence detection occurs at this step under the form of a detection confirmation;
4. Diffusion of a verdict message from the elected node through the spanning tree to finish the verification phase.

Some of those steps partially overlap in time. For example, when a node receives the verification message from one of its neighbors (the asking one), it forwards it to all its other neighbors (the replying ones) in the spanning tree (step 1) and, while waiting for their responses (step 3), it elaborates its own one (step 2) according to its local state. As soon as a negative response is detected on the node (either from itself or from a replying neighbor), the final response to the asking node can be sent. Otherwise, the node needs to wait for the gathering of all the responses from its replying neighbors before sending a positive response.

Finally, when the elected node has its own response and those of its neighbors, it deduces the final verdict, which corresponds to the actual global convergence detection when positive, and sends it to all of 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 that verdict.

As mentioned above, the response of each node depends on its state but also on its evolution during the verification phase. Indeed, in order to ensure that all the nodes are in local convergence at the same time (which corresponds to the criterion used in the sequential and synchronous versions), the response of a node is positive if and only if its residual stays under the threshold during the span of time between its last sending of a local convergence message (PartialCV) 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 evolution, the waiting of a pseudo-period is inserted before the sending of the response. 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 is fully representative of the actual evolution and state of that node until that time. In fact, those pseudo-periods form, at the global level of the system, a period which spreads from the global convergence detection to the end of the pseudo-period on the latest node.

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 (negative verdicts) of global detections, that tagging is also useful to distinguish the data messages related to different verification phases.

Finally, such a tagging system is also useful in the messages related to the global detection and verification processes in order to enhance the reactivity of the verification phase. Indeed, as mentioned above, each node is allowed to send a negative response as soon as it is able to deduce it, without waiting for all the responses of its replying neighbors. It is also the case for the elected node that will send a negative verdict without waiting for all the responses of its neighbors. However, those unused responses must be correctly managed when they finally arrive on a node and, in particular, they must not be confused with other responses related to a more recent verification phase, as they may consequently overlap.

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 Fig. 2 with four nodes linearly organized for a span of time beginning with the tag equal to k.

The whole mechanism of global detection and verification is detailed in Fig. 3, in the case of a global convergence detected and confirmed on node P_2. First of all, the processors reach local convergence and inform their adequate neighbor according to the leader election scheme, with PartialCV messages. Then, the global convergence is detected on node P_2 which initiates the verification phase by sending verification messages which are propagated through the system. As

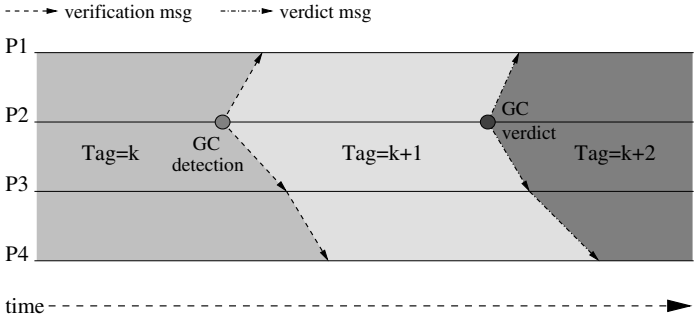

Fig. 2. Distinction of the successive phases during the iterative process

soon as the other nodes receive that message, they send their current version of local data to their neighbors with tagged data messages. And as soon as a node has received all its tagged data dependencies (not older than the last global detection) and has performed one iteration with them (dark grey blocks), it checks if its residual is still under the threshold since its last local convergence detection (light grey blocks) and sends the adequate response. As node P_3 is not elected and is not at an extremity of the system, it aggregates its response with the one of node P_4 and sends to its demanding node (P_2) the response corresponding to that sub-tree of the system relatively to the current root (P_2). The elected node P_2 meanwhile performs its own verification and as soon as it has finished it and has received all the other responses, it emits the verdict. The actual convergence detection takes place at this step when the verdict is positive. Finally, the verdict is sent and propagated through the system.

As can be seen, in case of a positive verdict, the whole process ensures that the residuals of all the nodes are under the threshold at least at the time at which the global convergence is detected on the elected node.

Concerning the correctness of the detection, we remind the reader that the ideal convergence criterion consists in verifying that the residual between two consecutive periods is small enough (under a convergence threshold). However,

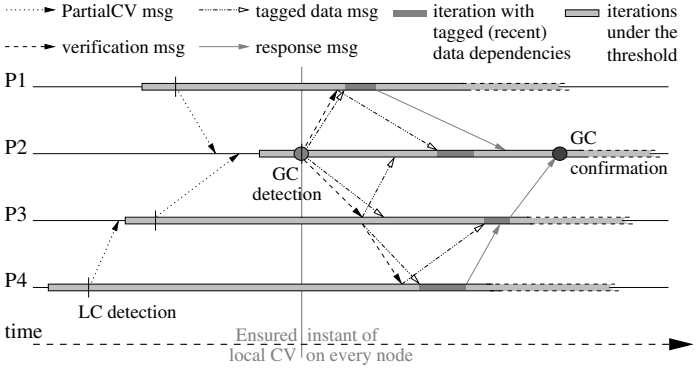

Fig. 3. Global convergence detection mechanism

as mentioned before, it is quite difficult and penalizing to identify all the periods at the global level during the process. But it is far simpler to explicitly trigger the execution of one period at a given time. This is what is done in Algorithm 9 in which the convergence criterion is composed of:

- A first pseudo-period with residual under the threshold;
- An arbitrary number (possibly 0) of consecutive pseudo-periods with residual under the threshold (the global convergence detection happens in that part);
- A last pseudo-period performed with data no older than the last global convergence detection and with residual under the threshold (the global convergence confirmation takes place at the end of that part).

The first two elements identify a global context of residual under the threshold. The last two elements contain an actual period which spreads between the global convergence detection and the global convergence confirmation. That period permits to ensure the validity of the detected convergence as it provides a similar stopping criterion as in the sequential and synchronous cases.

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 main states:

- **NORMAL:** the basic state during the 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 Fig. 4. Other states, present at inner levels (see Table 1), are not depicted here for clarity sake.

The final scheme obtained is given in Algorithm 9. Due to length constraints on that paper, only the list of the additional variables, according to the previous algorithm, is given in Table 1. The reader should refer to [1] for a description of the other variables.

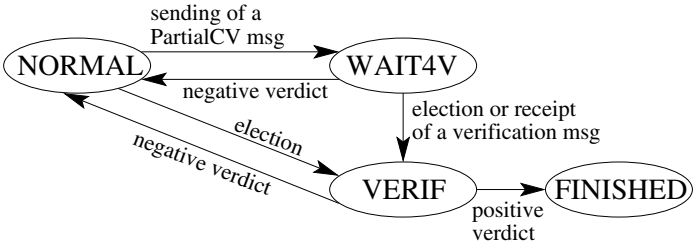

Fig. 4. State transitions in the global convergence detection process

Table 1. Description of the additional variables used in Algorithm 9

MyRank	unique identifier of the current node
State	current state of the node (NORMAL, WAIT4V, VERIF or FINISHED)
PhaseTag	identifier of the current phase on the node
PseudoPerBeg	boolean indicating that a pseudo-period has begun
PseudoPerEnd	boolean indicating the end of a pseudo-period
NbDep	number of computational dependencies of the 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

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

- **data message:**
  - identifier of the source node
  - iteration number on the source node at the sending time
  - phase tag of the source node at the sending time
  - data
- **PartialCV message** and **verification message:**
  - identifier of the source node
  - phase tag of the source node at the sending time
- **response message:**
  - identifier of the source node
  - phase tag of the source node 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.
- **InitializeVerif():** Initializes a new verification phase.
- **RecvDataDependency():** Manages the receipts of data dependencies. That function takes into account any newer data when the receiver is not in verification phase. 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, a mutual exclusion mechanism is used to ensure that only one node is elected.
- **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.
- **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 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 no other global convergence detection can happen before the end of the propagation of the verdict, there cannot be any confusion with a similar message coming from a previous verification phase.
- **ChooseLeader(integer, integer):** Chooses the elected node when there are two possible candidates whose identifiers are given in parameters. That function 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 1 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 2 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 3 Function InitializeVerif()
```
ReinitializePPer()
PhaseTag ← PhaseTag + 1
for Ind from 0 to NbNeig−1 do
  Resps[Ind] ← 0
end for
ResponseSent ← false
```

Algorithm 4 Function ReinitializePPer()
```
PseudoPerBeg ← false
PseudoPerEnd ← false
for Ind from 0 to NbDep−1 do
  NewerDep[Ind] ← false
end for
```

Algorithm 5 Function RecvDataDependency()
```
Extract SrcNode, SrcIter and SrcTag from the
message
SrcIndDep ← index of SrcNode in the list of
dependencies of the receiver (−1 if ∉)
if SrcIndDep ≥ 0 then
  if LastIter[SrcIndDep] < SrcIter and
  (State≠VERIF or SrcTag=PhaseTag) then
    Put the data from message at their place
    in the local data array used for the com-
    putations, according to SrcIndDep
    LastIter[SrcIndDep] ← SrcIter
    NewerDep[SrcIndDep] ← true
  end if
end if
```

Algorithm 6 Function RecvPartialCV()
```
Extract SrcNode and SrcTag from the message
SrcIndNeig ← index of SrcNode in the list of
neighbors of the receiver
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 7 Function RecvResponse()

```
Extract SrcNode, SrcTag and SrcResp from the message
SrcIndNeig ← index of SrcNode in the list of neighbors of the receiver
if SrcIndNeig ≥ 0
  and PhaseTag = SrcTag then
  Resps[SrcIndNeig] ← SrcResp
end if
```

Algorithm 8 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
```

Algorithm 9 Decentralized algorithm for the global convergence detection

```
for all P_i, i ∈ {1, ..., 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
    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
        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
            ResponseSent ← true
          else
            if PseudoPerEnd = true then
              if there remains only one 0 in Resps[] then
                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 ← true
              end if
            else
              if all the cells of NewerDep[] are true then
                PseudoPerEnd ← true
              end if
            ...
    end if
  until State = FINISHED
end for
```

6 Experiments

In order to evaluate the efficiency of our algorithm, we have compared it with our previous convergence detection algorithm on a typical asynchronous iterative algorithm based on the inverse power method. At each iteration of the algorithm, we solve a linear system using the multisplitting method [3]. A cluster of 16 machines (Pentium IV 3Ghz) with a 1Gbps network has been used. We have chosen the problem D of the CG problem, reported in [11], in which the matrix has a degree equals to 255,000 and 200 iterations are performed. That problem is very well suited to our comparison as it requires 200 convergence detections. We have implemented this algorithm in Java with the Jace environment [12]. In Table 2, we report the average times of ten executions of that problem with our new convergence detection algorithm and with our previous version. In the "without load" column, the machines run only our program without any other load. As can be seen, the execution times are very similar although slightly in favor of our new version. Also, in order to evaluate the robustness of our new algorithm, we have performed another series of experiments in the same conditions but we have slowed down some machines (2, 4 and 8) by adding an additional load on them. In order to obtain a correct convergence detection with our previous algorithm in such a context, some of its parameters, such

Table 2. Execution times with our two convergence detection algorithms and with or without external load

Version	Exec. times (s) without load	Exec. times (s) with 2 loads	Exec. times (s) with 4 loads	Exec. times (s) with 8 loads
Previous algorithm	661	740	772	866
New algorithm	655	712	732	809

as the number of successive iterations under the threshold and the maximal traversal time of the system, had to be increased. As can be seen in the last three columns of Table 2, such tunings imply larger detection latencies and thus worse execution times than our new version, which does not require any context-dependent tuning.

7 Conclusion

A new practical version of our decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms has been proposed. That new version presents the advantage of not requiring any information on the parallel system employed. This strongly broadens the contexts of use of AIAC algorithms since they are then surely and efficiently usable with large scale parallel systems, such as grids, in which the communication delays are subject to sharp variations and their upper bound is difficult to evaluate.

References

1. Bahi, J., Contassot-Vivier, S., Couturier, R., Vernier, F.: A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Transactions on Parallel and Distributed Systems 16, 4–13 (2005)
2. Saad, Y.: Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia (2003)
3. Bahi, J.M., Contassot-Vivier, S., Couturier, R.: Parallel Iterative Algorithms: from sequential to grid computing. Numerical Analysis & Scientific Computing Series. Chapman & Hall/CRC, Boca Raton (2007)
4. Maillard, N., Daoudi, E.M., Manneback, P., Roch, J.L.: Contrôle amorti des synchronisations pour le test d'arrêt des méthodes itératives. In: Renpar 14, Hamamet, Tunisie, pp. 177–182 (2002)
5. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Engelwood Cliffs (1989)
6. Savari, S.A., Bertsekas, D.P.: Finite termination of asynchronous iterative algorithms. Parallel Computing 22, 39–56 (1996)
7. Charão, A.S.: Multiprogrammation parallèle générique des méthodes de décomposition de domaine. PhD thesis, INPG (2001)

8. Bahi, J., Contassot-Vivier, S., Couturier, R.: Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. IEEE Transactions on Parallel and Distributed Systems 16, 289–299 (2005)
9. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Englewood Cliffs (1989)
10. Tarazi, M.E.: Some convergence results for asynchronous algorithms. Numer. Math. 39, 325–340 (1982)
11. The NAS parallel benchmark (1996), `science.nas.nasa.gov/Software/NPB/`
12. Bahi, J., Domas, S., Mazouzi, K.: Jace: a java environment for distributed asynchronous iterative computations. In: 12th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP 2004, Coruna, Spain, pp. 350–357. IEEE Computer Society Press, Los Alamitos (2004)