J Supercomput (2010) 53: 269–292
DOI 10.1007/s11227-009-0293-6

A decentralized and fault tolerant convergence detection algorithm for asynchronous iterative algorithms

Jean-Claude Charr · Raphaël Couturier · David Laiymani

Published online: 1 April 2009

Abstract This article presents an algorithm that performs a decentralized detection of the global convergence of parallel asynchronous iterative applications. This algorithm is fault tolerant. It runs a decentralized saving procedure which enables this algorithm, after a node's crash, to replace the dead node by a new one which will continue the computing task from the last check point. Combined with the advantages of the asynchronous iteration model, this method allows us to compute very large scale problems using highly volatile parallel architectures like Peer-to-Peer and distributed clusters architectures. We also present the implementation of this algorithm in the JaceP2P platform which is dedicated to designing and executing parallel asynchronous iterative applications in volatile environments. Numerous experiments show the robustness and the efficiency of our algorithm.

Keywords Decentralized global convergence detection mechanism · Peer-to-Peer environment · Distributed clusters · Fault tolerance

1 Introduction

Many natural and physical phenomena can be represented by numerical problems. Solving these problems allows scientists to numerically simulate these phenomena

J.-C. Charr · R. Couturier (✉) · D. Laiymani
Laboratory of computer science of Franche Comte, University of Franche-Comte,
IUT de Belfort-Montbéliard, Rue Engel Gros, BP 527, 90016 Belfort, France
e-mail: raphael.couturier@univ-fcomte.fr

J.-C. Charr
e-mail: jean-claude.charr@univ-fcomte.fr

D. Laiymani
e-mail: david.laiymani@univ-fcomte.fr

and to understand their behaviors and consequences. There are two classes of methods for solving numerical problems: direct or iterative methods. Direct methods give the exact solution of a problem after a finite number of operations (we can cite for instance: the *LU* or the Cholesky method). On the other hand, iterative methods compute many times the same block of operations until obtaining a good approximation of the solution (here we can cite the Jacobi or the conjugate gradient method). We then say that the method has converged to the solution of the problem. For some types of problems, iterative methods are more efficient than direct ones and they could be the only way to solve some problems. Most of the time, natural phenomena are represented by large and complex numerical problems that cannot be solved using a single regular computing unit. Therefore, the resolution methods must be parallelized in order to be simultaneously executed on many computing units. Iterative methods are easier to parallelize than direct ones and when dealing with very large problems; they are preferred over direct methods (especially if they give a good approximation of the solution in a small number of iterations).

These properties have led to the large expansion of parallel iterative algorithms. One of the major problems of parallelizing an iterative algorithm is how to detect its convergence. This mechanism is necessary to stop the iterative process in these methods. In sequential iterative algorithms, this consists in ensuring that the solution vector is stable and does not vary between two successive iterations. In practice, a residual value is computed (for example, $R = \max_i(|x_i^{k+1} - x_i^k|)$ where x_i^k denotes the value of the component i of array x at iteration k) and the system converges when R is smaller than a requested threshold.

The most common solutions for detecting the convergence of parallel iterative algorithms are both centralized and synchronous. For example, if a node detects that its subsystem has locally converged, it informs a server which is responsible for detecting if all the application's subsystems have converged. If so, the server declares that the application has globally converged. This centralized solution requires that all the nodes communicate with the server when they locally converge which can lead to a bottleneck if a large number of nodes locally converge at the same time. Moreover, this centralized solution is only compatible with parallel synchronous iterative algorithms. The parallel iterative algorithms can be classified in three main classes depending on how iterations and communications are managed. In the *Synchronous Iterations–Synchronous Communications (SISC)* model data are synchronously exchanged at the end of each iteration. All the computing units must begin the same iteration at the same time and important idle times are generated while waiting for other computing units to finish computing their iterations. The *Synchronous Iterations–Asynchronous Communications (SIAC)* model can be compared to the previous one except that data required on neighbors are sent asynchronously and without stopping current computations. This technique allows to partially overlap communications by computations, but unfortunately the overlapping is only partial and important idle times remain. Figure 1 illustrates two computing units executing a parallel iterative application implemented in the SIAC model. The grey blocks in Fig. 1 represent the computation phases, the white spaces represent the idle times and the arrows the asynchronous communications. It is clear that in a grid computing context, where the computational nodes are numerous, heterogeneous and widely distributed, the

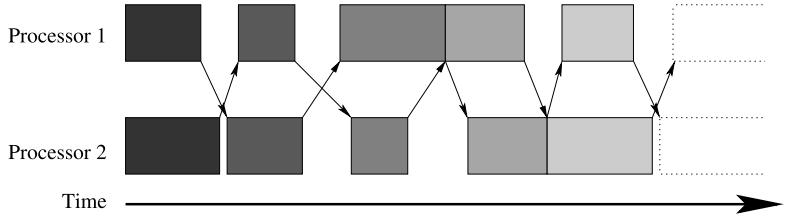

Fig. 1 Two processors computing in the Synchronous Iterations–Asynchronous Communications (SIAC) model

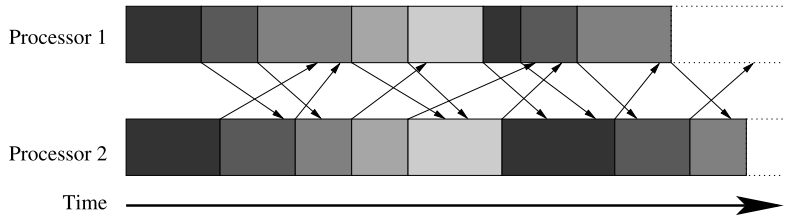

Fig. 2 Two processors computing in the Asynchronous Iterations–Asynchronous Communications (AIAC) model: the iterations and the communications are not synchronized

idle times generated by synchronizations are very penalizing. One way to overcome this problem is to use the *Asynchronous Iterations–Asynchronous Communications (AIAC)* model which is also known as the asynchronous iteration model [1]. Using this model, local computations do not need to wait to receive data from their neighbors. The computing units can perform their iterations using the data present at that time. Figure 2 shows that the idle times are totally suppressed between iterations. With this algorithmic model, the number of iterations required before the convergence is usually greater than for the two former classes. But as detailed in [2–4], the overall execution time is smaller if the numerical problem is being solved over heterogeneous and distant clusters. Since each computing unit can execute the next iteration without waiting for fresh data from neighbors, this model tolerates the loss of dependencies messages and moreover two nodes solving the same problem can execute different iterations at the same time. In Fig. 2, the first processor finishes the seventh iteration while the second processor is still executing the sixth iteration. These properties make this model unaffected by the crashes of computing nodes: if a node crashes, all the rest of the computing nodes may normally continue their tasks. The neighbors of the dead node do not have to wait for the reception of the dependencies messages from the dead node, and they can compute the next iterations using the last received dependency messages.

The centralized convergence detection mechanism, described above, requires a global synchronization at each recovery of the global state, which would indirectly synchronize the iterative process itself and would drastically reduce the ratio of asynchronism and its benefits. Moreover, in the asynchronous iteration model, the computing units do not necessarily receive new dependencies at each iteration. So, a node can compute many iterations without receiving any new data which could lead it to

a false local convergence detection. Therefore, the centralized convergence detection mechanism cannot be used with the asynchronous iteration mode. Nevertheless, this problem has been studied for a long time and the first global convergence detection method for asynchronous parallel algorithms was introduced in [5]. Then a new method that modifies the iterative process and is based on very strict assumptions was proposed by Savari and Bertsekas in [6]. Recently, a completely decentralized algorithm that does not modify the iterative process was introduced by Bahi et al. in [7, 8]. This algorithm, we called *Decentralized Convergence Detection* (DCD), is based on the "*Leader election*" algorithm [9, 10] and it is well suited to asynchronous computing. However, this algorithm is not fault tolerant.

This article has two objectives. First of all, it presents a fault tolerant decentralized algorithm which detects the global convergence of a parallel iterative asynchronous application. It is an evolution of the DCD algorithm in which a decentralized automatic saving mechanism was added. This allows a new node to replace a dead one by retrieving its last backup and continuing the calculation from that checkpoint without disturbing the other nodes or interrupting the application. Secondly, it illustrates the implementation of this algorithm into the JaceP2P platform [11]. This platform offers a design and execution environment specifically dedicated to AIAC algorithms. Nevertheless, in its current version, JaceP2P is only fault tolerant in the computing phase. *Integrating our algorithm into JaceP2P allows us to present a numerical parallel computing platform completely fault tolerant in both phases: the computing phase and the global convergence detection phase.* The experiments we conducted on the Grid'5000 platform with more than 700 cores, show that the computing and convergence detection processes in JaceP2P are now adequate for volatile environments and for a very small acceptable cost.

The rest of this paper is organized as follows. In Sect. 2, we present some related works, and in particular we explain the DCD algorithm. We also describe the different schemes used to make a parallel platform fault tolerant. Section 3 describes the old version of JaceP2P with the centralized global convergence detection mechanism. In Sect. 4, we introduce our distributed and fault tolerant algorithm and we present its implementation into JaceP2P. The results of the experiments are exposed in Sect. 5. They show the low influence of these modifications on the platform's performances and reveal the benefits of this decentralization. Finally, we end this paper with a conclusion and some research perspectives.

2 Related work

Global convergence detection has been widely studied (for example, in [12, 13]), however, most of the proposed algorithms are centralized and inadequate for large scale computing over distributed architectures. In general, the objective of the global convergence mechanism is to find out as soon as possible if all the computing nodes have locally converged, while assuming that all nodes are regularly updated. In a synchronous model, this task can be easily accomplished because at each iteration all the nodes are synchronized and receive fresh data from their neighbors. On the other hand, in an asynchronous iteration model, this operation is not trivial.

2.1 The DCD algorithm

This decentralized algorithm detects the global convergence of parallel iterative asynchronous applications. It is articulated around two phases:

- **Local convergence detection.** When the residue, locally evaluated, is smaller than the precision requested by the user, we say that the subsystem computed on that node has converged. However, the residue does not always decrease uniformly. Sometimes it oscillates around the threshold which can lead to a false detection of the global convergence. To our knowledge, there is no method that guarantees the local convergence of a node. However, it is possible to reduce the occurrence of false local convergence detections by using the *pseudo-period* concept. A *pseudo-period* can be defined as the smallest amount of time where a node receives at least one dependency from each of its neighbors. Therefore, a node converges locally if the residue stays under the threshold during one or many pseudo-periods.
- **Global convergence detection.** This phase is based on the *Leader election* algorithm which chooses a node dynamically so that it can execute a specified task (here, it is the detection of the global convergence). This method consists in transforming the graph of computing nodes into a tree. When a node converges locally, it checks the number of neighbors that did not converge (noted as s):
  - If $s > 1$, the node continues its iterative process.
  - If $s = 1$, it means that the node is a leaf in the tree or that all its neighbors but one have converged (that implies that its neighbors subtrees have also converged). In this case, the node sends a *convergence message* to its neighbor that did not converge yet. When the neighbor receives this message, it decrements by one its number of neighbors that did not converge.
  - If $s = 0$, it means that all the neighbors of the node (and all their subtrees) have converged. Therefore, it is the last node to converge locally and it is elected as *Leader* in order to detect the global convergence of the system.

After electing a Leader, there are four steps to perform in order to be sure that all the nodes are still in a local convergence state:

1. The Leader broadcasts a *verification message* to all its neighbors which also transmit the message to all their neighbors. In this way, the verification message is propagated to all the computing nodes and signals the beginning of the verification phase.
2. After receiving the verification message, each node begins a new pseudo-period in which it waits for new data from all its neighbors and it computes an iteration using these new dependencies. If, between the sending of the convergence message to the end of the pseudo-period, the residue does not ever cross over the threshold, the response to the verification phase is positive, otherwise it is negative. This phase allows the system to verify that the state of the nodes has evolved.
3. If a node develops a negative response, it directly sends it to the neighbor that sent him the verification message which propagates it to the Leader. On the other hand, if the response is positive, the node has to wait until its neighbors send him positive responses. Then it propagates the response to the Leader.

4. If the Leader receives or computes a negative response, it directly broadcasts a negative *verdict message* to all its neighbors which also propagates the verdict to all the nodes. When a node receives a negative verdict, it starts the whole global convergence mechanism all over again. On the other hand, if the Leader develops and receives from all its neighbors positive responses, it broadcasts a positive verdict to all the computing nodes. This positive verdict means that the system has globally converged. When a node receives a positive verdict, it ends its iterative process.

2.2 Fault tolerance

Fault tolerance is not a new research area, however, the emergence of large scale computing architectures makes it a vital subject to be developed. In fact, when using these types of architectures (distributed clusters or Peer-to-Peer architectures), the probability that a node fails is very high. If a node dies, the user may have to restart the application which may need dozens of computing hours using hundreds of nodes. One of the main advantages of AIAC algorithms is their tolerance to the loss of dependencies messages. This property makes it easier for these algorithms to develop fault tolerance policies for the computing processes than for other types of iterative algorithms. In this context, it seems very interesting to make the global convergence detection process also fault tolerant. This will provide an iterative algorithmic model well suited to computing environments with mainly volatile nodes.

Many schemes have been used to develop platforms that tolerate crashes. Each one has its advantages and drawbacks and choosing the right method highly depends on the characteristics of the targeted computing environment. Here are some common concepts:

- Checkpointing [14]: This is the most common scheme. It is based on saving for each node the essential data required to continue an application after a crash. This procedure is executed on precise checkpoints which are usually specified by the application's programmer. Nowadays, a good fault tolerant platform automatically detects the checkpoints without any intervention from the user; we say it provides transparent fault tolerance services. Checkpointing can be divided into two groups:

  1. Coordinated checkpointing [15]: When using this method, at each checkpoint, all the computing nodes in the system must be synchronized with a barrier call and a snapshot of all the system is taken and saved. If the system is synchronous and a node crashes, all the other nodes are blocked. Once the replacement node retrieves the last backup, all the nodes must also rollback to the last checkpoint and continue their tasks from that checkpoint. The coordinated checkpointing method cannot be used in an asynchronous model because at each checkpoint, it synchronizes the whole system which eliminates all the advantages of the asynchronous model.
  2. Uncoordinated checkpointing: Using this method, after a checkpoint, each node saves its data without any synchronization with the others. This allows the nodes to have backups at various times. This method suits the asynchronous models well.

Most of the fault tolerant platforms that use the checkpointing concept save their backups on a server. This requires having at least one reliable station. This centralization may cause a bottleneck. To reduce the effect of this problem, stations with high computing capacities and large bandwidth are used as backup servers. This method is not well adapted to distributed clusters and Peer-to-Peer environments where the nodes are volatile and have limited capacities. In such environments, the distributed redundant checkpointing method is used. In this method, the backups are saved on the computing nodes: each node saves its backups on its neighbors, so the number of backups per node is equal to the number of saving neighbors (which is specified by the user). This method does not make the platform fully tolerant to all types of crashes. Indeed, if a node and all its saving neighbors die at the same time, the backups for this node are lost and the platform cannot replace the dead node which will terminate the application. To make the platform more resistant to crashes, the user can increase the number of saving nodes, but, this could reduce the performances of the platform.

Finally, the backups saved either on a server or on computing nodes can be stored in the memory or in files (XML files, binaries, ...) on hard disks. If the size of the backups is too big, it is preferable to store them in files to prevent overloading the memory. However, this method slows down the platform because it has to access the hard disk every time a node demands a backup from its neighbors. As an example, the Open MPI [16] platform saves its backups in files on a local or distributed file system.

- Process Replication [17]: Using this scheme, each computing process is executed on many nodes. A master is chosen to represent each computing process and the others are considered as slaves. All the replications of a computing process execute the same process at the same time, but only the master communicates with the other masters. After each communication, it sends the information to its slaves which enables them to continue their tasks. If a master crashes, it is replaced by a slave. The dead node cannot be replaced by a new one because there is no backup to retrieve. So, after many crashes a computing process may be terminated if all its replications are dead. This method is not very efficient because if an application requires m machines, the platform has to reserve $n * m$ machines where n is the number of replicas per computing process. This method has been implemented in the P2P-MPI platform [18].
- Message logging [19]: using this scheme, each node saves the data that it receives on a log. When a node crashes, it is replaced by a new one which retrieves the log and executes from the beginning all the operations using the data saved in the log. This method saves the logs on a reliable server and requires to restart the computing process of the dead node from the beginning. If the application is large, the log may become full. For this reason, most of the time message logging is used in conjunction with the uncoordinated checkpointing method: Each node saves in a local log the messages it sent to its neighbors. When the log becomes too big, the node's data are saved on a reliable server. If a node crashes, the replacement node has to retrieve the backup from the backup server and the logs from its neighbors, then it executes all the operations that happened after the backup using the data stored on the logs. If a neighbor's log does not contain the required data, the neighbor must

retrieve its backup and rollback [20] to that state then it has to execute again all the operations executed after the backup. So, one crash may cause a domino effect and oblige many nodes to retrieve their old backups and rollback to that state. This method guarantees that the system will tolerate all the crashes that affect the computing nodes but it takes a lot of time to replace a dead node. This method is not adapted to volatile environments because it requires a reliable backup server which is impossible to have in such environments. This scheme has been implemented in the MPICH-V2 platform [21].

Since we are interested in executing asynchronous parallel iterative algorithms on Peer-to-Peer and distributed clusters architectures, we chose the uncoordinated checkpointing mechanism (on neighboring nodes) to make the decentralized convergence detection algorithm fault tolerant. It is the best method in these conditions. Nevertheless, it does not guarantee a 100% resistance but it drastically increases it: if an application uses 100 nodes with a 5% probability that a node crashes in a 100 seconds time interval, the probability that the system fails after 100 s is $5 * 100 = 500\%$. Now, if we applied the uncoordinated checkpointing on neighboring nodes and assumed that replacing a dead node takes 5 s and each node has five saving neighbors, the probability that the system fails after 100 s is $(5 * 5/100)^6 = 0.000244\%$ (equivalent to the probability that a node and its five saving neighbors die in the same five seconds time interval). So, the system resists to most of the crashes. In the rest of this article, we present a fault tolerant adaptation of the DCD algorithm and its implementation into the JaceP2P platform.

3 JaceP2P

The JaceP2P platform allows the development and the execution of AIAC algorithms that are implemented in Java (with/without dependencies between computing nodes) over volatile architectures. To tackle the specificities of AIAC algorithms, JaceP2P provides many essential mechanisms for these algorithms. In particular, it has an asynchronous communication mechanism and offers different threads for computing and communicating. The execution environment provided by JaceP2P allows computing nodes to connect and disconnect from the platform at any moment, in particular while executing a task. To resist to the volatility of the computing units, JaceP2P saves the state of each task at regular checkpoints.

3.1 Saving checkpoints

As mentioned before, the decentralized uncoordinated checkpointing is the best scheme for AIAC fault tolerant algorithms. It consists in saving automatically the state of a task at specific checkpoints on neighboring nodes. There are two drawbacks to this method. First of all, the process of checkpointing assigned to each node slows down the computing process because it runs in parallel with the computing process and consumes some of the node's computing power. Secondly, there is still a small risk that a backup could be lost if all backup neighbors die at the same time. It is possible to increase the robustness of the platform by increasing the number of backups

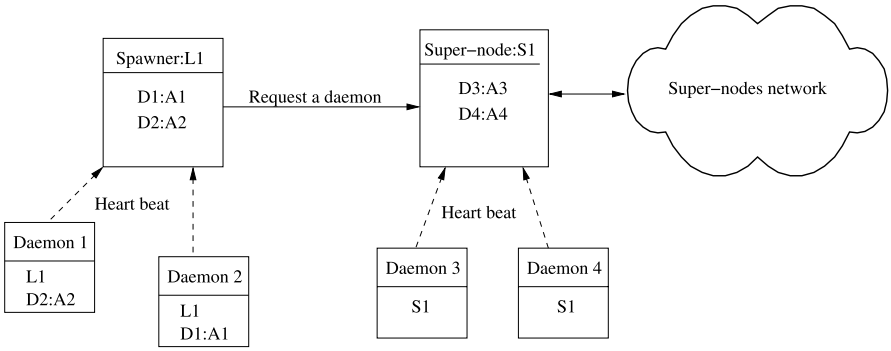

Fig. 3 The JaceP2P's architecture with 4 daemons. *D1* and *D2* are connected to the spawner which stores their respective addresses *A1* et *A2*

for each node, but this will slow down the computing process even further. A compromise between security and performance must be found. In JaceP2P, the computing data are saved every n iterations with n defined by the user as a parameter. If a node dies after saving its state, the platform retrieves the last backup and reruns this task on a new node. *The fundamental properties of AIAC algorithms make it possible for this node to continue the task from the last checkpoint while other computing units continue their tasks without having any concern for the dead node.*

3.2 Architecture

There are three types of nodes in JaceP2P (see Fig. 3):

1. **Super-node.** All the super-nodes form a network that stores in a distributed manner (in registers) the identifiers (name and IP address) of each daemon connected to the platform.
2. **Spawner.** The main role of a spawner is to launch the application. It needs to know the URL address of the server that contains the application's `.class` files, the number of nodes needed to solve the problem and all the eventual parameters of the application. All this information must be provided by the user. Once it has this information, the spawner contacts a super-node to reserve the nodes needed for this application (see next paragraph), and stores their identifiers in its register. This register is broadcasted to all the reserved nodes, so they can communicate with each others. When the application starts, the spawner becomes responsible for detecting the death of any computing node. If a daemon dies, the spawner contacts once again the super-node and reserves a new daemon which has to replace the dead one. The spawner modifies its register by adding the identifier of the new daemon and deleting that of the dead one. The new register is broadcasted to all the computing nodes (so they can communicate with the newly added node). The spawner also detects the global convergence of the parallel iterative application: when a daemon detects that its iterative task has converged, it notifies the spawner. If the spawner receives local convergence messages from all the daemons, it broadcasts a global convergence message to them. This message stops the computing processes and frees the daemons.

3. **Daemons.** The daemons execute the computing tasks forming the application. When a daemon is launched, it connects to a super-node to signal its availability. When it is reserved by a spawner, it retrieves from it a task and begins its execution. In order to be able to continue the task of a dead node, each daemon regularly saves its data on its neighbors. In practice, every n iterations each daemon sends its backup to a different neighbor according to the round-robin strategy. If a node is selected to replace a dead one, it contacts all its neighbors, retrieves from them the last backup and continues the computing task. When a daemon detects the local convergence of its iterative task, it signals it to the spawner. When a daemon receives a global convergence message, it begins computing the next step if there is one, otherwise it terminates its computing process and reconnects to a super-node.

4 A decentralized and fault tolerant mechanism for detecting the global convergence

4.1 The drawbacks of centralized global convergence detection mechanisms

In JaceP2P, the global convergence is centralized and is detected by the spawner. This centralization limits the scalability of the platform (the spawner may be overloaded when detecting the convergence of a problem solved by hundreds of nodes) and the global convergence currently implemented in JaceP2P does not necessarily reflect the state of all the nodes at the time of the detection. Indeed, in the AIAC algorithmic model, a node that executes many iterations without receiving new data from its neighbors, may locally converge (i.e., its residue is lower than the threshold) for a moment and signal it to the spawner, then diverge again. If the other daemons have also converged, the spawner will detect a false global convergence. Finally, in its current version, JaceP2P is vulnerable to crashes while detecting the global convergence. Here are some examples:

- If we assume that a daemon receives a global convergence message from the spawner and crashes just before saving it on its neighbors. The daemon replacing the dead one will retrieve an old backup and will not know that the parallel application has converged. If the application is composed of many steps, the node will continue to compute the old step indefinitely while the other nodes compute the next step. In this case, the application is blocked.
- If the spawner broadcasts a global convergence message to the daemons, and because of the network congestion, a node does not receive the message; this node will never know that the system has converged and that it must compute the next step. Thus, it will continue to compute the same step indefinitely and the application will be blocked forever.

To fix up these problems, we have implemented a fault tolerant version of the DCD algorithm. We used an hypercube topology in the implementation to reduce the broadcast time of the convergence messages. If n is the number of computing nodes, the maximum number of nodes that a message has to cross is equal to x with x equal to the smallest integer that verifies $n < 2^{x+1}$. In our implementation of the DCD algorithm, each node discovers its convergence neighbors using Algorithm 1.

Algorithm 1 Discovering convergence neighbors for an hypercube tree

```
1: {n = number of computing nodes}
2: {id = rank of the current computing node}
3: d ← 0
4: while 2^d < n do
5:     if id < 2^d and id + 2^d < n then
6:         id + 2^d is a convergence neighbor
7:     end if
8:     if id < 2^(d+1) and id > 2^d then
9:         id − 2^d is a convergence neighbor
10:    end if
11: end while
```

4.2 Critical procedure and backups

To be able to transform the DCD algorithm into a fault tolerant algorithm, we have defined a critical procedure as follows: "*each sequence of instructions that affects two neighbors and blocks one of them if that node crashes before saving its state*". Thus, in the DCD algorithm, if a node $n1$ locally converges and all its neighbors but one ($n2$) have also converged, $n1$ has to send a convergence message to $n2$. This message signals to $n2$ that $n1$ has converged. $n2$ decrements by one the number of its neighbors that did not converge yet and returns an acknowledge message to $n1$ which means that $n2$ has well received the convergence message. Once $n1$ receives the acknowledge message, it waits for the verification phase and stops sending convergence messages. Such a sequence of instructions is a critical procedure because it affects two neighbors ($n1$ and $n2$), modifies their states and if one of the two nodes dies during this procedure, the algorithm will be blocked indefinitely. In fact, if $n2$ dies or disconnects from the platform, the spawner detects that this node has not been sending any heartbeat message for a while, so it considers that it is dead. Then the spawner tries to replace this dead node: It contacts a super-node and requests an available node. The reserved node $n3$ replaces the dead one and to be able to continue the task, retrieves the last backup. If $n2$ died after returning the acknowledge message to $n1$, all the modifications, made after receiving the convergence message, are lost and $n3$ have retrieved an old backup with a false number of neighbors that have not converged yet. $n3$ will wait indefinitely for a convergence message from $n1$. To solve this problem, $n2$ must save its data before sending the acknowledge message to $n1$.

On the other hand, if $n1$ receives the acknowledge message and dies just before saving, the daemon replacing the dead node will retrieve an old backup. Thereafter, it detects again the local convergence of its task and sends a new convergence message to $n2$. The recipient notices that it has already received this message, so it ignores it but returns an acknowledge message to $n1$ so it can execute the next phase.

Identifying a critical procedure while detecting the global convergence is very difficult. The easiest approach is to save data after the execution of each instruction concerning the global convergence detection. This solution is not practical given its execution cost. Thus, it is necessary to determine the critical sequences of instructions and to minimize their numbers (see Sect. 4.4, for more information on the different

critical procedures in the DCD algorithm).

Now, two types of backups are implemented into JaceP2P:

- *Saving computing data*. It is executed every n iterations and once the node is sure of the local convergence of its subsystem after the verification phase. This backup contains the solution and dependencies vectors. Using this backup, a node can continue the task from that solution vector.
- *Saving convergence data*. It is executed during a critical procedure. It contains all the values necessary to detect the global convergence and maintains the system's coherence.

Distinguishing between these two types reduces the bad influence of checkpointing on the computing process. Actually, the size of the convergence backup is very small compared to the size of the computing backup. So, saving the convergence data again and again will not have a great impact on the performance of the computing process.

4.3 Acknowledge messages

The critical procedure concept, described in the previous paragraph, shows the importance of the acknowledge messages. It allows the sender to be sure that the recipient has received the convergence message. The DCD algorithm is composed of many phases. If a crash prevents a daemon from receiving a message that informs it that it has to pass to the next phase, the daemon and its subtrees nodes will be stuck in this phase while the others pass onto the next one. Therefore, we have to make sure that all the notifications concerning the global convergence detection are well received by their recipients. To accomplish that, we propose two solutions. First, the recipient treats the message and saves the results of that treatment on its neighbors. Then it returns the acknowledge message to the sender. Second, the recipient just saves the information sent in the message and returns the acknowledge message to the sender then handles the saved information afterward. Each approach has its advantages and drawbacks:

- Using the first solution, while the recipient treats the convergence message and saves the results on its neighbors, the sender remains blocked. However, the recipient has to save just once after processing the message's data.
- Using the second solution, the recipient saves the message's data and quickly returns the acknowledge message to the sender. However, after processing the message's data, it saves the results again on its neighbors.

To be able to choose between those two methods, we must evaluate the execution time cost for these two solutions. Unfortunately, it is very hard to achieve this test because it depends on many parameters like the network's latency, the processing speed of nodes and the number of components solved on each node. In practice, we can use the first method for the messages that do not require a lot of treatment (like local convergence messages) and the second method for the broadcasting messages because they could block the sender for a long time (while the message is propagated to all the subtrees nodes).

4.4 Overview of the fault tolerant algorithm

As described before, the decentralized convergence detection algorithm is decomposed into many phases, in this section, we will show how each phase has been modified in order to make the whole algorithm fault tolerant:

- **Before local convergence:** each node computes its iterative task and at the end of an iteration it asynchronously exchanges dependencies vectors with its neighbors then evaluates its residue in order to detect the local convergence. Throughout this phase, only data backups are saved on neighbors because in the asynchronous iteration model the loss of dependencies messages is tolerated and does not affect the convergence of the parallel iterative application (i.e., the solution of the studied problem).This phase corresponds to lines 5 to 9 in Algorithm 2.
- **After detecting the local convergence:** this is the first critical procedure. When a node detects that its residue is smaller than the requested precision, it computes a pseudo-period. It waits to receive new dependencies vectors from all its neighbors, then it computes a new iteration using these new data. If the residue evaluated after this iteration is still smaller than the threshold, the node declares its local convergence. This phase corresponds to lines 12 to 16 in Algorithm 3. As mentioned before in the DCD algorithm, the locally converged node tests the number of its unconverged neighbors (*nbNotConv*): if $nbNotConv = 1$, it executes the sequence of instructions illustrated in Fig. 4 and from lines 26 to 33 in Algorithm 3. If $nbNotConv = 0$ the node declares itself Leader and propagates the beginning of the verification phase on all its neighbors as in Fig. 5 and from lines 17 to 24 in Algorithm 3. If a node involved in this procedure fails, the node recovery mechanism described in Sect. 4.2 is applied.
- **Verification phase:** when a node receives a verification message it executes the sequence of instructions illustrated in Fig. 6 and in the ReceiveVerification procedure in Algorithm 4 where it begins the verification phase. During the verification phase, each node waits for new dependencies tagged with the verification tag. Afterward, it computes a new iteration using those new data. If during all these operations the residue is still under the threshold, the node elaborates a positive response, otherwise it elaborates a negative response.

Algorithm 2 Main()

```
1:  for each step do
2:      Initialize the global convergence variables
3:      globalConvergence ← false
4:      state ← NORMAL
5:      while globalConvergence = false do
6:          Compute an iteration
7:          if Residue < Threshold then
8:              pseudoConvergence ← true
9:          end if
10:         DetectGlobalConvergence(pseudoConvergence)
11:     end while
12: end for
```

Algorithm 3 DetectGlobalConvergence (pseudoConvergence) (1/2)

```
1: if action = SendVerification / action = SendVerdict then {To broadcast the verification
   and verdict messages}
2:    Broadcast a verification message / Broadcast the verdict
3:    Wait for an acknowledge message from each neighbor
4:    if received an acknowledge message from each neighbor then
5:       action ← nothing
6:       Save Convergence Data on neighbors
7:    else
8:       Broadcast the message on the next iteration
9:    end if
10: end if
11: if state = NORMAL and action = nothing then
12:    if pseudoConvergence = false then
13:       reinitialize the pseudo-period
14:    else
15:       if PseudoPeriod = true then
16:          localConvergence ← true
17:          if NbNeighboursNotConv = 0 then
18:             BroadcastVerification()
19:             Node ← Leader
20:             state ← Verification
21:             Wait for an acknowledge message from each neighbor
22:             if received an acknowledge message from each neighbor then
23:                Save Convergence Data on neighbors
24:             end if
25:          else
26:             if NbNeighboursNotConv = 1 then
27:                Send convergence message to the neighbor that did not converge
28:                Wait for an acknowledge message from the neighbor
29:                if received an acknowledge message from the neighbor then
30:                   state ← WAIT for Verification
31:                   Save Convergence Data on neighbors
32:                end if
33:             end if
34:          end if
35:       end if
36:    end if
37: else
38:    if state = WAIT for Verification then
39:       if pseudoConvergence = false then
40:          localConvergence ← false
41:          Save Convergence Data on neighbors
42:       end if
43:    else
44:       see that part on next page
45:    end if
46: end if
```

Algorithm 3 DetectGlobalConvergence (pseudoConvergence) (*Continued*) (2/2)

```
47: if state = NORMAL and action = nothing then
48:     see that part on previous page
49:     if state = WAIT for Verification then
50:         see that part previous page
51:         if state = Verification then
52:             if node = leader then
53:                 if pseudoConvergence = false or localConvergence = false or received a negative response then
54:                     Broadcast a negative verdict
55:                     Reinitialize the convergence variables
56:                     Wait for an acknowledge message from each neighbor
57:                     if received an acknowledge message from each neighbor then
58:                         Save Convergence Data on neighbors
59:                     end if
60:                 else
61:                     if PseudoPeriod = true and received positive responses from all neighbors then
62:                         Broadcast a positive verdict
63:                         state ← Finished
64:                         Wait for an acknowledge message from each neighbor
65:                         if received an acknowledge message from each neighbor then
66:                             Save Convergence Data on neighbors
67:                         end if
68:                     end if
69:                 end if
70:             else
71:                 if node did not send a response yet then
72:                     if pseudoConvergence = false or localConvergence = false or received a negative response then
73:                         if action = nothing then
74:                             Send a negative response to the neighbor that sent the verification message
75:                             Wait for an acknowledge message from the neighbor
76:                             if received an acknowledge message from the neighbor then
77:                                 Save Convergence Data on neighbors
78:                             end if
79:                         end if
80:                     else
81:                         if PseudoPeriod = true and received positive responses from all neighbors except one then
82:                             Send a positive response to the neighbor that did not send a response
83:                             Wait for an acknowledge message from the neighbor
84:                             if received an acknowledge message from the neighbor then
85:                                 Save Convergence and computing Data on neighbors
86:                             end if
87:                         end if
88:                     end if
89:                 end if
90:             end if
91:         end if
92:     else
93:         if state = FINISHED then
94:             globalConvergence ← true
95:         end if
96:     end if
97: end if
```

Algorithm 4 Reception methods

procedure: ReceiveConvergence()

```
1: if reloading = false and action=nothing then
2:     NbNeighboursNotConv ← NbNeighboursNotConv − 1
3:     Save convergence data on its neighbors
4:     Return an acknowledgment message to the sender
5: end if
```

procedure: ReceiveVerification()

```
1: if reloading = false then
2:     if state = WAIT for Verification then
3:         action ← SendVerification
4:         Initialize variables for verification phase
5:         state ← Verification
6:         Save convergence data on its neighbors
7:         Return an acknowledge message to the sender
8:     end if
9: end if
```

procedure: ReceiveResponse(Response)

```
1: if reloading = false then
2:     Store the response in the response vector
3:     Save convergence data on its neighbors
4:     Return an acknowledge message to the sender
5: end if
```

procedure: ReceiveVerdict(verdict)

```
1: if reloading = false then
2:     if verdict = true then
3:         state ← FINISHED
4:     else
5:         Reinitialize convergence variables
6:     end if
7:     action ← SendVerdict
8:     Save convergence data on its neighbors
9:     Return an acknowledge message to the sender
10: end if
```

- **Responding to the verification phase:** if a node elaborates or receives a negative message, it directly sends it to the sender of the verification message as explained in Algorithm 3 from lines 72 to 79. On the other hand, if it receives from all its neighbors (except from the sender of the verification message) positive responses and elaborates a positive response, it positively responds to the sender of the verification message. This case is presented in Algorithm 3 from lines 81 to 87 and sending a response is illustrated in Fig. 4. If the response is positive, the sender saves its computing and convergence data on its neighbors. This phase is also considered as a critical procedure. As above, if a node involved in this procedure fails, the node recovery mechanism described in Sect. 4.2 is applied.
- **Declaring a verdict:** when the Leader receives or elaborates a negative response, it broadcasts a negative verdict to all its neighbors as in Fig. 5 and in Algorithm 3

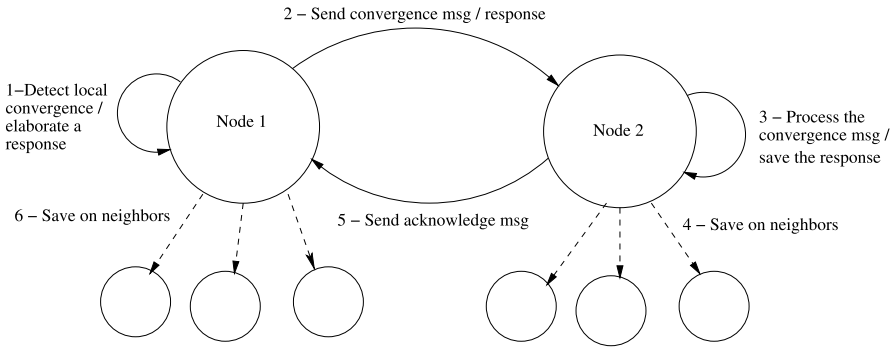

Fig. 4 The instructions executed when sending a convergence or a response message

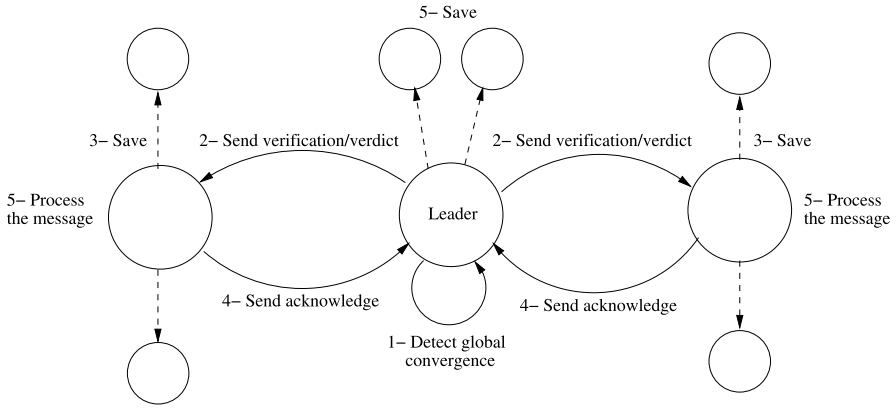

Fig. 5 The instructions executed when broadcasting a verification or a verdict message

from lines 53 to 60. Then it restarts the convergence detection algorithm all over again. When the Leader elaborates and receives from all its neighbors positive responses, it sets its state to global convergence and broadcasts a positive verdict to all its neighbors, as illustrated in Fig. 5 and in Algorithm 3 from lines 61 to 68. This is also a critical procedure and the node recovery mechanism described in Sect. 4.2 may be applied, if a node involved in this procedure fails.

- **Receiving a verdict:** when a node receives a verdict, it executes the sequence of instructions illustrated in Fig. 6 and in the ReceiveVerdict procedure in Algorithm 4. If the verdict is negative, it reinitializes its convergence variables and restarts the convergence detection algorithm for the same step. On the other hand, if the verdict is positive, it sets its state to global convergence and begins computing the next step or terminates the application. Here is the fourth critical procedure. If a node involved in it fails, the node recovery mechanism described in Sect. 4.2 is applied.

For more details on the algorithm, the reader can refer to the algorithms at the end of this paper.

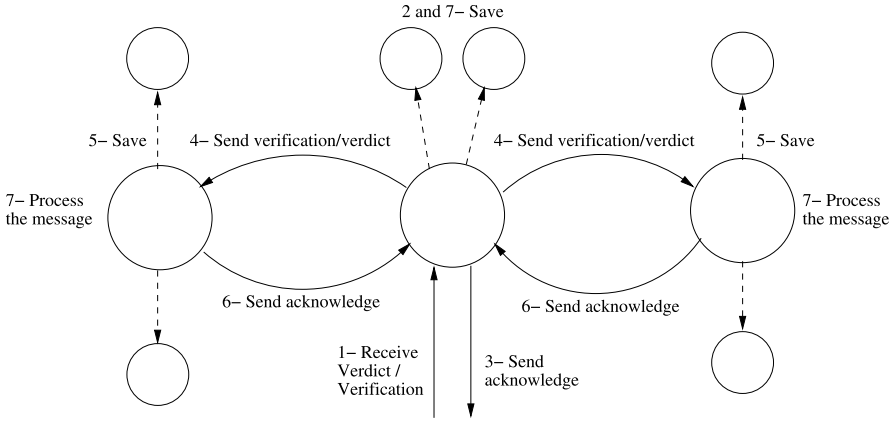

Fig. 6 The instructions executed when receiving a verification or a verdict message

5 Experiments

5.1 Mathematical description

In order to evaluate the robustness of our approach, we have implemented on JaceP2P a numerical iterative application that solves a three dimensional advection-diffusion equation. This equation represents mathematically the transport processes of pollutants, salinity, and so on, combined with their biochemical interactions. It follows an initial boundary value problem for a nonlinear system of PDEs, in which nonlinearity only comes from the biochemical interspecies reactions.

A system of 3D advection-diffusion-reaction equations has the following form:

$$\frac{\partial c}{\partial t} + A(c, a) = D(c, d) + R(c, t), \tag{1}$$

where c denotes the vector of unknown species concentrations, of length m, and the two vectors,

$$A(c, a) = [\mathbf{J}(c)] \times a^T, \tag{2}$$

$$D(c, d) = [\mathbf{J}(c)] \times d \times \nabla^T, \tag{3}$$

respectively, define the advection and diffusion processes ($\mathbf{J}(c)$ denotes the Jacobian of c with respect to (x, y, z)). The local fluid velocities u, v, and w of the field $a = (u, v, w)$ and the diffusion coefficients matrix d are supposed to be known in advance. A simulation of pollution evolution in shallow seas is obtained if a is provided by a hydro-dynamical model. The chemical species dynamic transport is defined by both advection and diffusion processes, whereas the term R includes interspecies chemical reactions and emissions or absorption from sources.

Following a common approach, $R(c, t)$ can be expressed using production and loss terms, denoted respectively by P and L:

$$R(c, t) = P(c, t) - L(c, t). \tag{4}$$

Both terms can be further refined:

$$P(c,t) = P_I(c,t) + P_S(t),$$
$$L(c,t) = (L_I(c) + L_S(t)) \times c.$$

While the terms P and L indexed by I denote the contributions (emission and absorption) from chemical interspecies reaction, the contributions from sources are indexed by S.

As we consider a test problem in three spatial dimensions, reduced to two chemical species, the system described by (1) becomes a system of two PDEs:

$$\begin{pmatrix} \frac{\partial c^1}{\partial t} \\ \frac{\partial c^2}{\partial t} \end{pmatrix} + \begin{pmatrix} \nabla c^1 \times a \\ \nabla c^2 \times a \end{pmatrix} = \begin{pmatrix} \nabla \cdot ((\nabla c^1) \times d) \\ \nabla \cdot ((\nabla c^2) \times d) \end{pmatrix} \begin{pmatrix} R^1(c,t) \\ R^2(c,t) \end{pmatrix}, \tag{5}$$

where matrix d satisfies

$$d = \begin{pmatrix} \epsilon(x) & 0 & 0 \\ 0 & \epsilon(y) & 0 \\ 0 & 0 & \epsilon(z) \end{pmatrix}, \tag{6}$$

and the vector function $R(c,t)$ is defined according to:

- production terms P_I and P_S

$$P_I(c,t) = \begin{pmatrix} q_4(t)c^2 \\ q_1 c^1 c^3 \end{pmatrix}, \quad P_S(t) = \begin{pmatrix} 2q_3(t)c^3 \\ 0 \end{pmatrix}, \tag{7}$$

- loss terms L_I and L_S

$$L_I(c) = \begin{pmatrix} 0 & q_2 c^1 \\ q_2 c^2 & 0 \end{pmatrix}, \quad L_S(t) = \begin{pmatrix} q_1 c^3 & 0 \\ 0 & q_4(t) \end{pmatrix}. \tag{8}$$

Clearly, the coupling of the two PDEs is induced by the reaction term R.

As far as transport is concerned, in this work, vertical advection (dimension z) is neglected, and the velocity field vector a is supposed to have a constant value. Hence, a is given by

$$a = (u, v, w) = (-V, -V, 0), \tag{9}$$

with $V = 10^{-3}$. Regarding diffusion coefficients, horizontally they are positive constants, whereas the vertical ones vary. Thus, we have a matrix d of the form

$$d = \begin{pmatrix} K_h & 0 & 0 \\ 0 & K_h & 0 \\ 0 & 0 & K_v(z) \end{pmatrix}, \tag{10}$$

where $K_h = 4.0 \times 10^{-6}$ and $K_v(z) = 10^{-8} \times \exp(\frac{z}{5})$.

For the reaction term (apart from the two unknown concentrations of the contaminants, i.e., c^1 and c^2), the different quantities in (5)–(8), are chosen as follows:

- $c^3 = 3.7 \times 10^{16}$, $q_1 = 1.63 \times 10^{-16}$ and $q_2 = 4.66 \times 10^{-16}$,
- $q_3(t), q_4(t)$ are chosen according to ($j = 3, 4$),

$$\begin{aligned} q_j(t) &= \exp[\tfrac{-a_j}{\sin(\omega t)}] \quad \text{if } \sin(\omega t) > 0, \\ q_j(t) &= 0 \qquad\qquad\qquad\quad \text{if } \sin(\omega t) \leq 0, \end{aligned} \tag{11}$$

using the following parameters: $\omega = \pi/43200$, $a_3 = 22.62$ and $a_4 = 7.601$.

For more information concerning the mathematical description of the problem, the reader can refer, for example, to [23]. On the other hand, for more information concerning the implementation of the problem, we can cite [22].

To solve this problem we used the Multisplitting-Newton method [24]: the principle is to split the initial domain into several subdomains in order to assign one of them to each computing unit involved in the parallel computation. At each iteration, each subdomain is solved sequentially using the standard Newton method. All the nodes send their boundaries values asynchronously to their neighbors at the end of an iteration. Then the fault tolerant DCD algorithm is executed in order to detect if the parallel iterative application has globally converged.

5.2 Experiments' conditions and results

The primary tests were conducted on a small local cluster containing just 20 nodes in order to be sure of the stability of the platform. After this phase and in order to prove that our modifications increased the scalability of JaceP2P, we have run many experiments on the Grid'5000 platform [25] using hundreds of nodes. As shown in Fig. 7, Grid'5000 is a public national experimental Grid, geographically distributed on nine sites in France. It is composed of many heterogeneous clusters. The connection's bandwidth between two sites is 10 Gb/s and between two nodes on the same site is about 1 Gb/s. In our experiments, the node's crashes were produced by a `shell` script that kills n nodes every m seconds.

Experiment n° 1: one site

In this experiment, 252 biprocessor machines (so, in total about 500 processors) located on a single site (e.g., Orsay) were used to solve a system containing 405.224.000 components which simulates a 90 seconds time interval. The nodes used are equipped with 2 AMD Opteron 246 2.0 GHz processors or with 2 AMD Opteron 250 2.4 GHz processors. Initially, we considered evaluating the cost of our modifications on the performance of JaceP2P. However, we quickly noticed that it is almost impossible to accurately measure this cost on this platform because our experiments test an heterogeneous application (the computing load is different between two distinct computing nodes) using heterogeneous nodes and executing an asynchronous application. Furthermore, the platform uses a decentralized mechanism to detect the global convergence of the system and for two successive executions, JaceP2P may

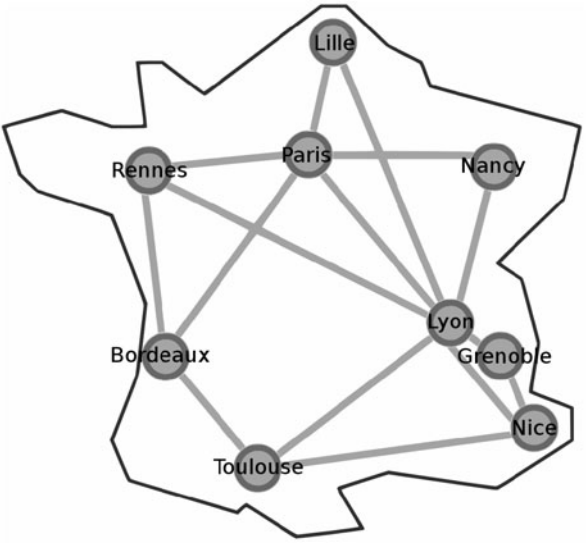

Fig. 7 The sites of Grid'5000 in France

Table 1 Execution time with 3 random crashes every n seconds

n	∞	90	60	50	25
Execution time	522 s	873 s	1003 s	1135 s	1738 s
Total number of crashes	0	30	51	68	205

order the same node to solve two distinct parts of the system. These characteristics make it almost impossible to obtain similar execution times when running the same application twice with the same initial conditions.

For this reason, we chose to test JaceP2P while changing the number of crashes during a simulation. In Table 1, we present the results of these experiments. We can notice that the platform has resisted to the huge amount of crashes. New daemons have replaced the dead nodes and thanks to the decentralized backups, they were able to continue their tasks from the last backup. JaceP2P seems to be more scalable than before (*the old version of JaceP2P with a centralized global convergence detection was not able to use a huge number of nodes like in this experiment because the spawner was quickly overloaded with convergence messages*) and the overprice due to crashes is quite acceptable.

Experiment n° 2: three distant sites

In order to simulate a Peer-to-Peer environment (huge number of nodes, heterogeneous and volatile nodes, heterogeneous networks ...), we have executed the same application over 3 distant clusters:

- Site 1 (Nancy), where each machine is equipped with 2 double cores 1.6 GHz Intel Xeon 5110.
- Site 2 (Sophia), where each machine is equipped with 2 processors AMD Opteron 246 2.0 GHz.

Table 2 Execution time while randomly killing a node every n seconds on each site

n	∞	90	60	50
Execution time	565	1438	2008	2050 s
Total number of crashes	0	48	100	122

- Site 3 (Orsay), described in the first experiment.

Therefore, 252 stations, with more than 700 cores, are used to solve a system containing 405.224.000 components which simulates a 90 second time interval. Table 2 presents the results of the experiments.

We noticed that while computing on distant clusters, the platform suffers more from crashes than when computing on a single cluster. Indeed, the communications between nodes on distant clusters are slower than between nodes on a single cluster which implies that the nodes need more time to retrieve backups from their neighbors and to broadcast convergence messages. As the previous one, this experiment proves that our modifications to the JaceP2P platform have made it resistant to crashes without significantly reducing its performance.

6 Conclusions and perspectives

In this article, we have presented a decentralized global convergence detection algorithm that works in a volatile environment. To our knowledge, this algorithm is the first of its kind. It is an extension of the Bahi et al. algorithm based on uncoordinated checkpointing and acknowledge messages. This new version of the algorithm has been implemented into the JaceP2P platform. The experiments conducted on the Grid'5000 platform showed that JaceP2P is now more stable and more scalable than before and that the backups do not significantly reduce the performance of the platform. In this way, this algorithm can be used in a grid computing environment where crashes are very frequent.

Our current and future works concern the decentralization of the detection of dead nodes (now this task is assigned to the spawner) and the attribution of a specific register to each node based on its needs. At the moment, all the nodes have the same register containing all the nodes addresses. We would like each node to have a register that only contains the addresses of its computing, saving, and convergence detection neighbors. These modifications will increase the scalability of the platform and reduce the bottlenecks and network congestions. Finally, we would like to duplicate the spawner in order to make it fault tolerant.

References

1. Bahi J, Contassot-Vivier S, Couturier R (2002) Asynchronism for iterative algorithms in a global computing environment. In: The 16th annual int symp on high performance computing systems and applications (HPCS'2002), June 2002, Moncton, Canada, pp 90–97
2. Bahi JM, Contassot-Vivier S, Couturier R (2006) Performance comparison of parallel programming environments for implementing AIAC algorithms. J Supercomput 35:227–244

3. Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation: numerical methods. Prentice Hall, Englewood Cliffs
4. Vuillemin P (2006) Calcul itératif asynchrone sur infrastructure pair-à-pair : la plate-forme JaceP2P. Thèse, Université de Franche Comté
5. Bertsekas DP, Tsitsiklis JN (1989) Convergence rate and termination of asynchronous iterative algorithms. In: 1989 Int Conf on Supercomputing, Crete, Greece. ACM SIGA RCH, 1989, pp 461–470
6. Savari SA, Bertsekas DP (1996) Finite termination of asynchronous iterative algorithms. Parallel Comput 22:39–56
7. Bahi JM, Contassot-Vivier S, Couturier R (2007) Parallel iterative algorithms: from sequential to grid computing. Numerical analysis & scientific computing series. Chapman Hall/CRC, London
8. Bahi J, Contassot-Vivier S, Couturier R, Vernier F (2005) A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans Parallel Distrib Syst 16(1):4–13
9. El-Ruby M, Kenevan J, Carison R, Khalil K (1991) Leader election in distributed computing systems. In: Proceedings of computing in the 90's, 1991. LNCS, vol 507. Springer, Berlin, pp 350–356.
10. Antonoiu G, Srimani PK (1996) A self-stabilizing leader election algorithm for tree graphs. J Parallel Distrib Comput 34(2):227–232
11. Bahi J, Couturier R, Vuillemin P (2006) JaceP2P: an environment for asynchronous computations on Peer-to-Peer networks. In: 2006 IEEE int conf on cluster computing (Cluster 2006), 2006. IEEE Computer Society Press, Los Alamitos
12. Dijkstra EW, Feijin WHJ, van Gasteren AJM (1983) Derivation of a termination detection algorithm for distributed computation. Inf Process Lett 16:217–219
13. Francez N (1980) Distributed termination. ACM Trans Program Languages Syst 2:42–55
14. Plank JS, Beck M, Kingsley G, Li K (1995) Libckpt: transparent checkpointing under UNIX. USENIX Winter, pp 213–224
15. Cao G, Singhal M (1998) On coordinated checkpointing in distributed systems. IEEE Trans Parallel Distrib Syst 9:1213–1225
16. Hursey J, Squyres JM, Mattox T, Lumsdaine A (2007) The design and implementation of checkpoint/restart process fault tolerance for open MPI. In: IPDPS 2007—the 21st IEEE international parallel distributed processing symposium, Long Beach, California, USA, 26 March 2007
17. Elnozahy EN, Zwaenepoel W (1992) Replicated distributed process in Manetho. In: The twenty-second international symposium on fault-tolerant computing, Boston, USA, 1992. IEEE Computer Society, Los Alamitos, pp 18–27
18. Genaud S, Rattanapoka C (2005) A Peer-to-Peer framework for robust execution of message passing parallel programs on grids. In: Recent advances in parallel virtual machine and message passing interface, 12th European PVM/MPI users' group meeting, Sorrento, Italy, September 18–21, 2005, pp 276–284
19. Alvisi L, Marzullo K (1995) Message logging: pessimistic, optimistic, and causal. In: Proceedings of the 15th international conference on distributed computing systems, Vancouver, British Columbia, Canada, May 30–June 2, 1995. IEEE Computer Society Press, Los Alamitos
20. Elnozahy EN, Zwaenepoel W (1992) Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit. IEEE Trans Comput 41:526–531
21. Bouteiller A, Cappello F, Herault T, Krawezik G, Lemarinier P, Magniette F (2003) MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: SC2003: igniting innovation, Phoenix, AZ, November 15–21, 2003. ACM Press, IEEE Computer Society Press, New York, Los Alamitos
22. Byrne GD, Hindmarsh AC (1998) User documentation for PVODE an ODE solver for parallel computers. Tech rep UCRL-ID-130884. Lawrence Livermore National Laboratory, Livermore, CA
23. Verwer JG, Blom JG, Hundsdorfer W (1996) An implicit-explicit approach for atmospheric transport-chemistry problems. Appl Numer Math 20:191–209
24. Bahi J, Miellou J-C, Rhofir K (1997) Asynchronous multisplitting methods for nonlinear fixed point problems. Numer Algorithms 15:315–345
25. www.grid5000.fr (2009)

Jean-Claude Charr received his Master degree in Computer Science at the University of Paul Sabatier (France) and the Lebanese University in 2006. Since 2006, he is a Ph.D. student at the University of Franche-Comte (France). His main research interests lie in the parallel computing domain such as grid and peer-to-peer computing, asynchronism and distributed virtual reality.

Raphaël Couturier is Professor in Computer Science at the University of Franche-Comte, France. Professor Couturier also serves as Head of the Computing Science Department of IUT Belfort-Montbéliard. He received his Ph.D. in 2000 from the University Henri Poincaré, France. His research interests include parallel and distributed computation, numerical algorithms and data mining.

David Laiymani has obtained his Ph.D. degree in 1997 from the University of Franche-Comté, where he is actually Assistant Professor. He is working in the Distributed Numerical Algorithms team in the Computer Science Laboratory (LIFC) and his research interests include parallel computing and grid middleware.