JACEP2P-V2: A Fully Decentralized and Fault Tolerant Environment for Executing Parallel Iterative Asynchronous Applications on Volatile Distributed Architectures

Jean-Claude Charr, Raphaël Couturier, and David Laiymani

Laboratory of computer sciences, University of Franche-Comté (LIFC)
IUT de Belfort-Montbéliard, Rue Engel Gros, BP 527, 90016 Belfort, France
Tel: +33-3-84587781
{jean-claude.charr,raphael.couturier,david.laiymani}@univ-fcomte.fr

Abstract. This article presents JACEP2P-V2, a Java environment dedicated to designing parallel iterative asynchronous algorithms (with direct communications between nodes) and executing them on global computing architectures or distributed clusters composed by a large number of volatile heterogeneous distant computing nodes. This platform is fault tolerant, multi-threaded and completely decentralized. In this paper, we describe the different components of JACEP2P-V2 and the various mechanisms used for scalability and fault tolerance purposes. We also evaluate the performance of this platform and we compare it to JACEP2P by implementing a parallel iterative asynchronous application and by executing it on a volatile distributed architecture using both platforms.

Keywords: Decentralized global Convergence, Peer-to-Peer architectures, Distributed clusters, Parallel iterative asynchronous algorithms.

1 Introduction

The simulation of natural and nuclear reactions (like climate change or nuclear fusion) requires solving very large and complex numerical problems and necessitates computing very large data in order to obtain precise and reliable results. These problems cannot be solved using a single computing unit because most of the time, the numerical problems are so huge that a single computing unit does not have enough memory nor computing power to store the application and to solve it. These problems could only be solved using simultaneously many computing resources. With the development of new reliable network equipments and the emergence of cheap and fast desktops, scientists are able nowadays to create distributed architectures using only these simple low cost devices. Most of the time, these distributed architectures tends to replace equally powerful but more expensive supercomputers. In numerical computing we can distinguish three kinds of distributed architectures:

N. Abdennadher and D. Petcu (Eds.): GPC 2009, LNCS 5529, pp. 446–458, 2009.

1. **Local clusters** are composed of similar workstations connected via a local network with low latency and large bandwidth.
2. **Distributed clusters** are composed of many distant clusters with heterogeneous computing units that are connected via heterogeneous networks with high latency and large bandwidth.
3. **Global computing architectures** are mainly composed of public unused heterogeneous workstations connected to Internet. These architectures offer free and unlimited computing power but suffer from the volatility of the nodes and from the slowness of communications.

Most of the time, the local cluster architecture does not have enough computing power to solve very large numerical problems. Therefore, in this paper, we are only interested in distributed clusters and global computing architectures. Using one of these parallel architectures, developers have to parallelize the method that solves the numerical problem in order to execute a subsystem of the problem on each computing unit. However, to use these architectures, the developer needs to manage carefully the exchange of data between the different computing units, especially when using high latency networks with heterogeneous and volatile nodes.

There are two classes of methods to solve numerical problems:

- **Direct methods** give the exact solution of a numerical problem after executing a finite number of operations. However, they are not really suited to distributed clusters and global computing architectures because they require several synchronizations.
- **Iterative methods** iterate many times the same block of instructions until obtaining a good approximation of the solution (e.g., Jacobi or Conjugate Gradient algorithms [1]). An iterative method converges when the "residual vector" (there is many methods to evaluate the value of the residue, for example $Residue = max_i(|x_i^{k+1} - x_i^k|)$ where x_i^k denotes the value of the component i at iteration k) is inferior to the precision (ϵ) requested by the user. Iterative methods are well adapted for very large problems.

Since we would like to solve very large numerical problems, in the rest of this paper we only focus on iterative methods. Now, from a parallel point of view, there are two models of parallel iterative algorithms:

- **The synchronous iteration model**. Using this model, as shown in figure 1, after each iteration (represented by a filled rectangle in the figure), a node sends its dependencies to its neighbors and waits for the reception of all the dependency messages from all its neighbors. Then all the nodes must synchronize to test if the system has globally converged. This results in large periods of idle time (represented by white spaces between the rectangles). These synchronizations can drastically penalize the overall performances in the case of large scale heterogeneous platforms. Moreover, if a dependency message is lost, the receiver will wait forever for that message and the application will be blocked. In the same way, if a computing node is dead,

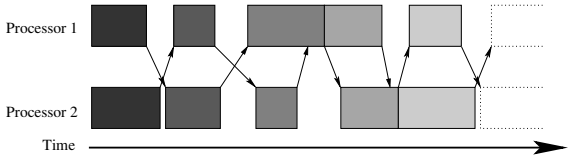

Fig. 1. Two processors using the synchronous iteration model

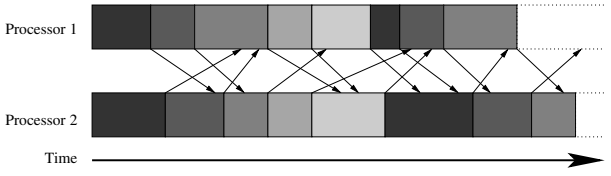

Fig. 2. Two processors using the asynchronous iteration model

all the rest of the nodes will be blocked until the dead node is replaced. In conclusion, this model is not well suited for large scale volatile computing environments.

– **The asynchronous iteration model** [2]. Using this model, as shown in figure 2, after each iteration, a computing node sends its dependencies to its neighbors and begins the next iteration using the last received dependency data. The node does not have to wait for the reception of the dependencies messages from its neighbors, consequently, there is no idle time anymore (no white spaces between iterations). The sending and the receiving mechanisms are asynchronous and the computing nodes tolerate the loss of data messages. Even if a node dies, the rest of the nodes can continue the computation process using the last dependency message sent by the dead node. In conclusion, the asynchronous iteration model is well adapted for volatile environments like peer-to-peer architectures or distributed clusters.

To tackle the specificities of the asynchronous iteration model on distributed clusters or global computing architectures, we have developed JACEP2P-V2, a new and improved version of JACEP2P [3]. JACEP2P-V2 is a fully decentralized and fault tolerant platform dedicated to designing and executing parallel iterative asynchronous algorithms on volatile architectures. The aim of this paper is to present the design and the features of this new platform.

The rest of this paper is organized as follows: in the next section we present some existing platforms related to our work. These platforms are briefly described and the differences between our work and these platforms are emphasized. In the third section, we present JACEP2P's architecture and its limits. Then, we introduce JACEP2P-V2 and we describe in details its mechanisms and functionalities. In the fourth section, we present the experiments conducted on the Grid5000[4] testbed using JACEP2P-V2 to solve a numerical problem. Finally, we end this paper with a conclusion and some perspectives.

2 Related Work

Recently, many middlewares for distributed clusters and global computing platforms have been developed. However, most of them are not well adapted for large numerical computing. Here are some examples:

- **Seti@home [5]:** The amazing success that this platform has achieved, helped the creation of generalized environments like Xtrem Web [6] and Boinc [7]. They are independent of the application and fault tolerant. The user creates a parallel application and executes it using the "workers". However, in these platforms, the clients cannot communicate with each others. So they cannot execute a parallel computing application with dependencies between nodes.
- **JXTA [8]:** It is an open-source project, composed of a set of peer-to-peer protocols that allows any connected device (cell phone to PDA, PC to server) on the network to communicate and collaborate. However, JXTA is a low level platform and offers a lot of general functionalities that are not well adapted for executing complex computing applications.
- **ProActive [9]:** It is an Open Source Java library for parallel, distributed, and multi-threaded computing. Although this environment provides direct communications between nodes using the RMI technology, when two nodes communicate, they must be synchronized (even if the concept of future objects exists). Moreover ProActive uses a global checkpointing mechanism [10] that requires synchronizing all the nodes in case of failures. In consequence, the asynchronous iteration model cannot simply be used on this platform.
- **JACE [11]:** "Java Asynchronous Computation Environment" is a multithre- aded Java based library designed to build asynchronous iterative algorithms and execute them in a Grid environment. In JACE, two nodes exchange data (synchronously or asynchronously) using either Sockets, RMI or NIO (New Input/Output). However, this platform is not fault tolerant, so it cannot be used in large scale volatile environments.

3 JACEP2P-V2

3.1 JACEP2P

JACEP2P is a distributed platform implemented using the Java programming language and dedicated to developing and executing parallel iterative asynchronous applications. JACEP2P executes parallel iterative asynchronous applications with dependencies between computing nodes. On the other hand, JACEP2P is fault tolerant which allows it to execute parallel applications over volatile environments and even for stable environments like local clusters and grids, it offers a safer and crash free platform.

JACEP2P's architecture. Figure 3, presents the architecture of JACEP2P and the various components that form the platform:

– The first entity is the "super-node" (represented by a big circle in figure 3). Each super-node stores in its register the identifiers (IP address) of all the computing nodes that are connected to it and are not executing an application. The super-node regularly receives heartbeat messages (represented by doted lines in figure 3) from the computing nodes connected to it. If the super-node does not receive a heartbeat message from a computing node included in its register for a given period of time, it declares that this computing node is dead and deletes its identifier from the register.
– The second entity is the "spawner" (represented by a square in figure 3). When a user wants to execute a parallel application, he or she launches a spawner with the required parameters which contacts a super-node to reserve the required computing nodes. The super-node reserves the demanded daemons (see next paragraph) which are removed from the super-node's register and returns to the spawner a register containing the identifiers of the reserved computing nodes. When the spawner receives the register, it creates a task for each computing node and starts the execution of the tasks on the respective daemons. The spawner has to send its register to all the computing nodes in order for them to be able to communicate with each others. Moreover, the spawner is responsible for detecting the disconnection of a computing node. Indeed, when the computing nodes are reserved by the spawner, they start sending their heartbeat messages to the spawner. If the spawner detects that a computing node has not sent to it a heartbeat message for a while, it declares that this computing node is dead. Then, it fetches a new one from the super-node in order to replace the dead one. The spawner initializes the new daemon, which retrieves the last backup of the dead node and continues the computing task from that checkpoint. Finally, the spawner is also responsible for detecting the global convergence of the parallel iterative application. When a subsystem converges locally, the computing node executing it sends a convergence message to the spawner. If the spawner receives a convergence message from all the computing nodes, it declares that the parallel iterative application has globally converged.
– The third entity is the "daemon" or the computing node (represented in figure 3 by a hashed small circle if it is free and by a white small circle if it is executing an application). Once launched, it connects to a super-node and

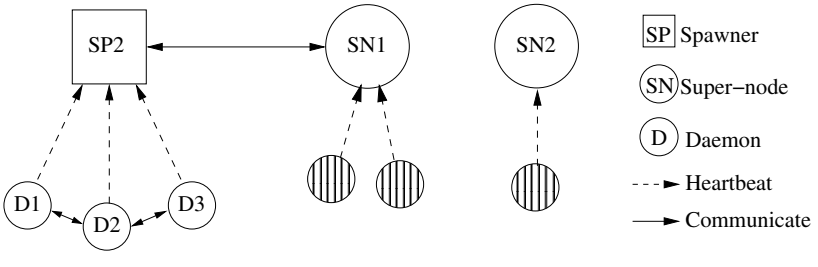

Fig. 3. JACEP2P's architecture and the different components

waits for a task to execute. During the execution of the parallel application, the daemons can communicate with each others and they regularly save their state on their neighbors. At the end of a task, the daemons reconnect to the super-node.

To be able to execute asynchronous iterative applications, JACEP2P has an asynchronous messaging mechanism (for more details interested readers can refer to [3]) and to resist to daemons' failures, it implements a distributed backup mechanism called the uncoordinated distributed transparent checkpointing [12]. This method allows daemons to save their data on neighboring daemons without any user intervention. The asynchronous nature of the application allows two daemons to execute two different iterations, thus each daemon saves its status without synchronizing with other daemons. This decentralized procedure allows the platform to be very scalable, with no weak points and does not require a secure and stable station for backups. Moreover, if a daemon dies, the other computing nodes continue their tasks and they are not affected by this failure.

JACEP2P's limitations. In [13], the experiments' results proved that the first version of JACEP2P performs very well and presents a relatively small overhead. Nevertheless, this version has some important limits:

- JACEP2P is not fully fault tolerant. Indeed, in this version, spawners' crashes are not tolerated. Moreover, while executing the global convergence process, the platform does not resist well to the disconnection of daemons.
- JACEP2P has a centralized failure detection. The spawner receives heartbeat messages from all the daemons and detects if a daemon is dead. If the application is being executed by a large number of daemons, the spawner will be overloaded with heartbeat messages. This will delay the detection of a dead daemon and could even lead to a false crash detection. Moreover, if many daemons die successively and there is only one spawner to handle the dead daemons, then the spawner will take a lot of time to replace them. This may reduce the performance of the platform.
- JACEP2P has a centralized global convergence detection mechanism which is not well adapted for executing asynchronous parallel iterative algorithms on volatile architectures. The daemons executing such applications do not receive dependencies messages from their neighbors at each iteration. This may lead to a false local convergence and thus result to false global convergence detection. Furthermore, the spawner could be overloaded by convergence messages, if many daemons converge locally at the same time.
- JACEP2P has many centralized mechanisms like launching the application, detecting the global convergence and detecting the dead nodes. These centralizations limit the scalability of JACEP2P.
- In JACEP2P, each daemon receives the whole register which contains the identifiers of all the daemons executing the application. If a daemon crashes and is replaced by a new one, the spawner has to notify the modifications to all the daemons in order to update their registers. This could overload the spawner and increase the congestion of messages in the network.

To remedy these problems we present JACEP2P-V2 a fully decentralized and fault tolerant platform. In the next subsection, we will describe in details the functionalities and characteristics of the new platform.

3.2 JACEP2P-V2's Architecture

Figure 4 shows the architecture of JACEP2P-V2 where we notice that there are two spawners handling the execution of a single application and each group of entities (spawners, daemons and super-nodes) forms a circular network. JACEP2P-V2 has similar entities as JACEP2P but with different functionalities:

- **Super-nodes.** They form a circular network now and store in an equally distributed manner the identifiers of all the computing nodes that are connected to the platform and that are not executing any application. Each super-node has a status table containing the number of connected computing nodes to each super-node and all the super-nodes share a "token" that is passed successively from a super-node to the next one. Once a super-node has the token, it computes the average number of computing nodes connected to a super-node (avg) using the status table. If avg is lower than the number of computing nodes connected to it, then it sends the identifiers of the extra computing nodes to the super-nodes that have the number of computing nodes connected to them less than avg. If the number of computing nodes connected to it has changed, it broadcasts the information to all the super-nodes in the platform. Finally, it passes the token to the next super node. This distribution reduces the overload of the super-nodes.
- **Spawners.** When a user wants to execute a parallel application that requires N computing nodes, he or she launches a spawner. The spawner contacts a super-node to reserve the N computing nodes plus some extra nodes in order to transform them into spawners. When the spawner receives the register from the super-node, it transforms the extra daemons into spawners and stores the identifiers of the rest of the daemons in its own register. Once

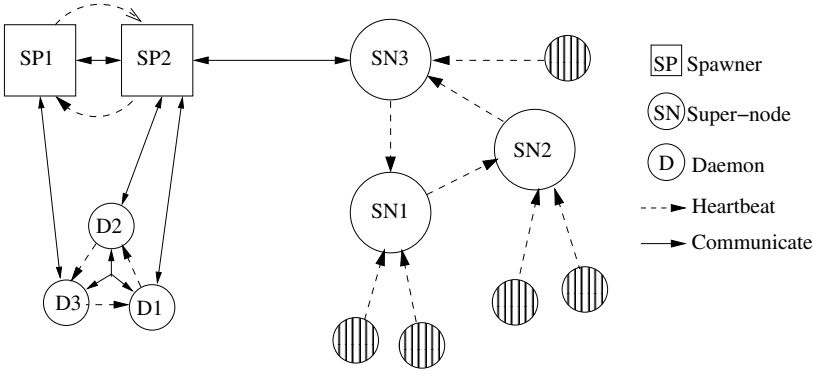

Fig. 4. JACEP2P-V2's architecture and different components

the extra nodes are transformed into spawners, they form a circular network and they receive the register containing the identifiers of the computing nodes. Then each spawner becomes responsible for a subgroup of computing nodes, starts the tasks on the computing nodes under its command and sends a specified register to them. So each computing node receives a specified register that only contains the identifiers of the daemons it interacts with and that depends on the application being executed. These specified registers reduce the number of messages sent by the spawners to update the register of the daemons after a daemon crashes because usually a small number of daemons is affected by this crash.
- **Daemons.** Once they begin executing an application they form a circular network which is only used in the failure detection mechanism. Each daemon can communicate directly with the daemons whose identifiers it has in its register.

3.3 JACEP2P-V2 Functionalities and Characteristics

After describing quickly the modifications made on the architecture of JACEP2P-V2, in this section we present in details the different new functionalities and characteristics implemented in JACEP2P-V2:

- **Completely decentralized.** JACEP2P-V2 is completely decentralized. In fact, all the tasks are divided between the entities of the same type. For example, a daemon can be connected to any super-node and the group of super nodes shares equally the control of the free daemons connected to the super-node network. The spawners are also decentralized: once a spawner is launched to execute an application, it quickly duplicates itself into several spawners (depending on the number of daemons required to execute the parallel application) by transforming some daemons into spawners. Each spawner becomes responsible for starting the application on a subgroup of daemons and handling the needs of that subgroup. For the computing nodes, each one executes a part of the application and the sum of their work gives the solution of the global problem. This distribution of tasks, allows JACEP2P-V2 to solve very large problems and thus to become very scalable with theoretically no limiting conditions.
- **Completely fault tolerant.** We have implemented many mechanisms to make the three entities that form the core of JACEP2P-V2 fault tolerant. An important concept available for the three entities is the decentralized crash detection mechanism. It enables the neighbors of a node to detect if it is dead or alive. Each group of entities forms a circular network. This organization is needed to apply the decentralized crash detection mechanism we have implemented. Each entity has a "heartbeat thread" that signals regularly to the next node in the circular network that the sender is still alive and another thread, the "scan thread" , that tests at each iteration if the previous node in the circular network has recently sent a heartbeat message. If for a given period of time the node does not receive a heartbeat message from

the previous one, the scan thread detects that the previous node is probably dead. Depending on the type of the dead node, the disconnection is handled. In fact, each entity has a restoring mechanism which is also dependent on the saving mechanism used for each type of entity.

- For daemons we use the distributed backup mechanism described before and we have implemented two types of backup in JACEP2P-V2. The first backup contains all the information concerning the state of a node (convergence data) and its computing process (solution vector). This backup is saved each N iterations (N given by the user and usually depends of the length of an iteration) on a different neighbor using the "round-robin" strategy. On the other hand, the second backup only contains the status data. This backup has a smaller size and it is saved when the status of a daemon has changed, especially when it concerns the global convergence detection mechanism. This backup is saved on all the backup neighbors simultaneously. Once a daemon detects that the previous daemon is dead, the daemon signals it to the spawner responsible for it. The spawner contacts a super-node and acquires a new daemon. The new daemon replaces the dead one and retrieves the last status backup and the last data backup. Once it has the backups, it continues the task from that last checkpoint. During all this operation, all the other daemons continue their tasks normally.
- For spawners, we use the duplication mechanism. The spawner is duplicated into many spawners. All the spawners have all the information concerning all the daemons and each one manages only a subgroup of daemons. If a spawner dies, the next spawner detects it (using the same scheme described before). Then, that spawner contacts a super-node, gets a new daemon and transforms it into a spawner. Once it becomes a spawner, it receives the register containing the identifiers of all the daemons executing the application, it identifies its subgroup of daemons, it informs them that it is the new spawner and it is reintegrated into the circular spawner network.
- For super-nodes, there is no saving mechanism, they do not contain very valuable information. When a dead super-node is detected by the next super-node, it is rejected from the circular super-node network. All the daemons that were connected to the dead super-node will reconnect onto another super-node.

– **Multi-threaded.** JACEP2P-V2 is multi-threaded. The computing process is never blocked by the exchange of data messages between daemons. Each functionality (communicating, detecting crashes, saving and computing) has its own thread.
– **Decentralized global convergence detection algorithm:** JACEP2P-V2 has implemented this algorithm to detect efficiently the global convergence of the asynchronous iterative parallel algorithms executed on the platform. It consists of two phases: the detection phase and the verification phase. This algorithm is presented in details in these papers [2,13].

– **Acknowledge mechanism:** JACEP2P-V2 tolerates the loss of data messages when it executes parallel asynchronous iterative algorithms. However, it has to ensure the right reception of the convergence messages by the receivers in order to ensure a coherent system. Therefore, we had to implement an acknowledge mechanism dedicated to the convergence messages. When a computing node sends a convergence message to another one, the receiver handles the received message (this usually changes the state of the receiver) then it saves its new state on its backup neighbors and next it returns an acknowledge message to the sender. Once the sender receives the acknowledge message, it knows that the receiver has received the convergence message and then it saves its state on its backup neighbors so it does not send the convergence message again. On the other hand, if the sender does not receive the acknowledge message, it knows that the message did not reach its destination and that it has to send the convergence message all over again.

4 Experiments

In order to evaluate the benefits of the improvements that have been implemented in JACEP2P-V2, we have conducted two sets of experiments on Grid'5000 French national grid. The same experiments were realized using JACEP2P and JACEP2P-V2 in order to compare both platforms. Both sets of experiments were realized on two different architectures. During these tests, both platforms had to execute a parallel iterative application that solves a three dimensional advection-diffusion equations system. This system represents mathematically the transport processes of pollutants, salinity, and so on, combined with their bio-chemical interactions.

4.1 Mathematical Description

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 $A(c, a) = [\mathbf{J}(c)] * a^T$ and $D(c, d) = [\mathbf{J}(c)] * d * \nabla^T$ 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. For more details, readers can refer to [14].

Table 1. Execution time with 3 random crashes every n seconds

n	∞	90	60	30
Execution time for JACEP2P	522s	873s	1003s	1611s
Total number of crashes for JACEP2P	0	30	51	159
Execution time for JACEP2P-V2	495s	565s	595s	744s
Total number of crashes for JACEP2P-V2	0	18	28	68

4.2 First Experiment: Local Cluster

In this experiment, we compare JACEP2P to JACEP2P-V2 while executing the same application on a single site. This application solves a system containing 405.224.000 components and that simulates a 90 seconds time interval. 252 bi-processors computing units, located in Orsay, were used to run this application. The computing nodes were equipped with 2 AMD Opteron 246 2.0GHz or 250 2.4GHz processors. To prove that the two platforms are fault tolerant, we used a shell script that randomly kills three computing nodes each n seconds.

The results for this set of experiments are presented in table1. It shows the execution times taken by JACEP2P and JACEP2P-V2 to solve the problem with various frequencies of nodes crashes. It is obvious that JACEP2P-V2 outperforms JACEP2P in each category. We also notice that JACEP2P-V2 is less affected than JACEP2P by the disconnection of computing nodes. Indeed, when the computing nodes disconnect frequently, JACEP2P suffers a lot because of the centralized nature of some of its components. On the other hand, with the JACEP2P-V2's decentralized dead nodes detection, the dead nodes are detected faster by their neighbors and thus they are replaced quickly by new ones to continue their tasks. These mechanisms reduce the influence of the crashes on the performance of JACEP2P-V2 platform.

4.3 2nd Experiment: Distributed Clusters

In this second set of experiments, we aimed at simulating a global computing architecture which has the following characteristics: large number of heterogeneous computing units, high latency communications and volatile nodes. So, we used the same number of computing nodes but this time we have chosen them from three distant sites in order to have heterogeneous computing nodes. Moreover, the latency between two nodes from distinct sites is superior to the one between two nodes located on the same site, thus the latency of the communications is also heterogeneous. The computing nodes were selected from the following sites: Nancy where each station is equipped with 2 double cores 1.6 GHz Intel Xeon 5110, Sophia where each station is equipped with 2 processors AMD Opteron 246 2.0GHz and Orsay which is described in the first experiment. We executed the same application as in the first experiment using JACEP2P and JACEP2P-V2. We have also simulated the volatility of the computing nodes by using the same perturbator script. However in this experiment, the script killed one daemon on each site each n seconds. The results for this set of experiments are presented in

Table 2. Execution time with one crash every n seconds at each site

n	∞	90	60	50
Execution time for JACEP2P	565s	1438s	2008s	2050s
Total number of crashes JACEP2P	0	48	100	122
Execution time for JACEP2P-V2	581s	624s	632s	663s
Total number of crashes JACEP2P-V2	0	19	30	38

table 2. As in the previous experiment, JACEP2P-V2 outperforms JACEP2P, in particular when the environment is highly volatile. Moreover, the crashes overhead is totally acceptable in JACEP2P-V2. These experiments prove that the modifications implemented in JACEP2P improve its performance on volatile architectures that suffer from high latency between computing nodes.

5 Conclusion and Perspectives

In this paper we have presented the new version of JACEP2P, called JACEP2P-V2. This parallel platform is dedicated for designing and executing parallel asynchronous iterative applications in volatile environments. This new version is fully fault tolerant which makes it able to resist the failure of any node in the platform, especially the ones executing an application. We have also implemented a decentralized mechanism for detecting dead nodes in order to replace them. We also conducted two sets of experiments using two different architectures. In all these tests JACEP2P-V2 outperformed JACEP2P.

In the near future, we want to test JACEP2P-V2 using more computing units. We also would like to test it on a real global computing architecture, using unused public computing units connected via Internet. Finally, we want to implement many types of iterative asynchronous applications on JACEP2P-V2 to show the benefits of this platform and its general utility.

References

1. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS publishing (1996)
2. Bahi, J., Contassot-Vivier, S., Couturier, R.: Parallel Iterative Algorithms: from sequential to grid computing. Numerical Analysis & Scientific Computating, vol. 1. Chapman & Hall/CRC, Boca Raton (2007)
3. Bahi, J., Couturier, R., Vuillemin, P.: JACEP2P: an environment for asynchronous computations on peer-to-peer networks. In: Cluster 2006, pp. 1–10 (2006)
4. grid 5000, http://grid5000.fr
5. Seti@home, http://www.setiathome.berkeley.edu/
6. Cappello, F., Djilali, S., Fedak, G., Herault, T., Magniette, F., Néri, V., Lodygensky, O.: Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. Future Generation Computer Systems 21(3), 417–437 (2005)
7. BOINC, http://www.boinc.berkley.edu/
8. JXTA, http://www.jxta.org/

9. ProActive, http://www.proactive.inria.fr/
10. Cao, G., Singhal, M.: On coordinated checkpointing in distributed systems. IEEE Transactions on PDS-9 (12), 1213–1225 (1998)
11. Bahi, J., Domas, S., Mazouzi, K.: Jace: a java environment for distributed asynchronous iterative computations. In: PDP 2004, Spain, February 2004, pp. 350–357 (2004)
12. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under UNIX. In: USENIX Winter, pp. 213–224 (1995)
13. Charr, J.C., Couturier, R., Laiymani, D.: A decentralized convergence detection algorithm for asynchronous iterative algorithms in volatile environments (submitted, 2008)
14. Bahi, J., Couturier, R., Mazouzi, K., Salomon, M.: Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. Applied Mathematical Modelling 30(7), 616–628 (2006)