JaceV: a Programming and Execution Environment for Asynchronous Iterative Computations on Volatile Nodes

Jacques M. Bahi, Raphaël Couturier, and Philippe Vuillemin

LIFC, University of Franche-Comté, France*** †
{jacques.bahi,raphael.couturier,philippe.vuillemin}@iut-bm.univ-fcomte.fr,
WWW home page: http://info.iut-bm.univ-fcomte.fr/and/

Abstract. In this paper we present JaceV, a multi-threaded Java based library designed to build asynchronous parallel iterative applications (with direct communications between computation nodes) and execute them in a volatile environment. We describe the components of the system and evaluate the performance of JaceV with the implementation and execution of an iterative application with volatile nodes.

Key words: Asynchronous iterative algorithms, computational science problems, desktop grid computing, volatile nodes.

1 Introduction

Nowadays, PCs and workstations are becoming increasingly powerful and communication networks are more and more stable and efficient. This leads scientists to compute large scientific problems on virtual parallel machines (a set of networked computers to simulate a supercomputer) rather than on expensive supercomputers. However, as the node count increases, the reliability of the parallel system decreases. As a consequence, failures in the computing framework make it more difficult to complete long-running jobs. Thus, several environments have been proposed to compute scientific applications on volatile nodes using cycle stealing concepts. In this paper, we consider as volatile node any volunteer personal computer connected to a network (WAN or LAN[1]) that can be used as a computational resource during its idle times. The aim of this work is to run scientific computations in such a volatile framework.

In this paper, we are interested in iterative algorithms. Those algorithms are usually employed for sparse systems (like some linear systems) or when direct methods cannot be applied to solve scientific problems (e.g. for polynomial root finders). In the parallel execution of iterative algorithms, communications must be performed between computation nodes after each iteration in order to satisfy

*** Candidate to the Best Student Paper Award

† this work was supported by the "Conseil Régional de Franche-Comté"

[1] World Area Network or Local Area Network

all the computing dependencies. For that reason, the reliability of the system is a very important feature in such a context and can become a limiting factor for scalability. Hence, it is necessary to study this reliability according to the different classes of parallelism. We consider three concepts (or classes) of parallelism with different characteristics.

1. Grid computing environments enable the sharing and aggregation of a wide variety of geographically distributed computational resources (such as supercomputers, computing clusters...). In such architectures, communications are very fast and efficient and the topology of the system is quite stable.
2. Desktop grid computing environments (also called Global Computing) exploit unused resources in the Intranet environments and across the Internet (e.g. the SETI@home project [2]). In this class of parallelism, the architecture is fully centralized (client-server-based communications), tasks are independent and the topology of the system is completely dynamic (nodes appear and disappear during the computation).
3. Peer-To-Peer (P2P) environments are networks in which each workstation has equivalent capabilities and responsibilities. The architecture is completely decentralized (peers directly communicate between each other) and the topology of the system is completely dynamic.

As reliability is generally ensured in a Grid computing context, we do not consider this class; furthermore, several frameworks are already available to implement and run parallel iterative applications in such environments. Concerning Desktop grid, although this class can provide much more resources than the first one, it is generally not directly suitable for parallel iterative computations as long as communication is restricted to the master-slave model of parallelism.

For that reason we would like to gather functionalities and characteristics of the latter two cases: 1) a centralized architecture to manage all the nodes of the system akin to a Desktop grid environment with volatile nodes and 2) direct communications between computation nodes like in P2P environments. The purpose of this paper is to describe a programming environment allowing users to implement and run parallel asynchronous iterative algorithms on volatile nodes. Asynchronous algorithms can be used in a significant set of applications. Indeed, scientific applications are often described by systems of differential equations which lead, after discretization, to linear systems $Ax = b$ where A is a M-matrix (i.e. $A_{ii} > 0$ and $A_{ij} \leq 0$ and A is nonsingular with $A^{-1} \geq 0$). A convergent weak regular splitting can be derived from any M-matrix and any iterative algorithm based on this multiplitting converges asynchronousely (see [1, 4, 10] and the references therein).

As idle times and synchronizations are suppressed in the asynchronous iteration model (i.e. a computing node can continue to execute its task without waiting for its neighbor results), we do believe this solution is the most suitable in an environment with volatile nodes. Furthermore, computations formulated in parallel asynchronous iterative algorithms are much less sensitive to heterogeneity of communication and computational power than conventional synchronous parallel iterative algorithms.

We do not consider the synchronous iteration model because it is neither convenient for this volatile framework, nor for the heterogeneity and scalability.

In this paper, we describe JaceV, a multi-threaded Java based library designed to build asynchronous parallel iterative applications (with direct communications between computation nodes) and execute them in a desktop grid environment with volatile nodes. To the best of our knowledge, this work is the first one presenting a volatile execution environment with direct communications between computing nodes and allowing the development of actual scientific applications with interdependent tasks.

The following section presents a survey of desktop grid and volatility tolerant environments. Section 3 presents the architecture of JaceV and an overview of all its components. Section 4 describes the scientific application implemented with JaceV (the Poisson problem) in order to perform experiments. Section 5 evaluates the performance of JaceV by executing the application in different contexts with volatile nodes. In section 6, we conclude and some perspectives are given.

2 Related work

Cycle stealing in a LAN environment has already been studied in the Condor [9] and Atlas [3] projects. However, the context of LAN and the Internet are drastically different. In particular, scheduling techniques [7, 8] need to be adapted for a Global Computing environment due to: 1) the very different communication and computing performance of the targeted hosts, 2) the sporadic Internet connection and 3) the high frequency of faulty machines.

MPICH-V and MPICH-V2 [11, 13] (message passing APIs[2] for automatic Volatility tolerant MPI environment) have been proposed for volatile nodes. However, MPI is not a multi-threaded environment. As a consequence, it is not suitable for asynchronous iterations in which it is convenient to separate communications and computation.

XtremWeb [16] is a Desktop Grid and Global Computing middleware which allows users to build their own parallel applications and uses cycle stealing. However, this environment does not provide direct communications between the different computing nodes of the system. As a consequence, it is not suitable for implementing and running parallel iterative applications.

Ninflet [6] is a Java-based global computing system. It is designed to overcome the limitations of Ninf [5] that currently lacks security features as well as task migration. The goal of Ninflet is to become a new generation of concurrent object-oriented system which harnesses abundant idle computing powers, and also integrates global as well as local network parallel computing. Unfortunately, as with the XtremWeb environment, Ninflet only applies Master-Worker pattern and does not provide direct communications between computation nodes.

In [15], no environment is proposed but the authors define the requirements for an effective execution of iterative computations requiring communication on

[2] Application Programming Interfaces

a desktop grid context. They propose a combination of a P2P communication model, an algorithmic approach (asynchronous iterations) and a programming model. Finally, they give some very preliminary results from application of the extended desktop grid for computation of Google pagerank and solution of a small linear system.

Jace [14] is a multi-threaded Java based library designed to build asynchronous iterative algorithms and execute them in a Grid environment. In Jace, communications are directly performed between computation nodes (in a synchronous or an asynchronous way) using the message passing paradigm implemented with Java RMI[3]. However, this environment is not designed to run applications on volatile nodes.

3 The JaceV system

3.1 The goal of JaceV

As described in the previous section, Jace is fully suitable for running parallel iterative applications (in a synchronous or asynchronous mode) in a Grid computing context where nodes do not disappear during computations. Then, it was essential to completely redesign the Jace environment in order to make it tolerant to volatility. To do this, it is necessary to develop a strategy to periodically save the results computed by each node during the execution in order to restart computations from a consistent global state [12] when faults occur.

We propose JaceV, the volatility tolerant implementation of Jace (JaceV for Jace Volatile). JaceV allows users to implement iterative applications and run them over several volatile nodes using the asynchronous iteration model and direct communications between processors.

Hence, when a computer is not used during a defined finite time, it should automatically contribute to compute data of a parallel iterative application already running (or to be started) on the system. *A contrario*, when a user needs to work on this workstation, the resource must instantaneously be freed and this node must automatically be removed from the system. In this way, a volatility tolerant system must both tolerate appearance and disappearance of computation nodes without disturbing the final results of the applications running on it. In fact, JaceV tolerates N simultaneous faults (N being the number of computational resources involved in an application) without disturbing the results at all.

3.2 Architecture of the system

A JaceV application is a set of *Task* objects running on several computation nodes. Like in Jace, the different Task objects of an application cooperate by exchanging messages and data to solve a single problem. The JaceV architecture

[3] Remote Method Invocation

consists of three entities which are JVMs [4] communicating with each others: 1) the *Daemons*, 2) the *Spawner* and 3) the *Dispatcher*. Since JaceV is based on Jace, all the communications performed between the different entities of the system are based on Java RMI and threads are used to overlap communications by computations during each iteration.

The user of the JaceV system could play two types of roles, one being the *resource provider* (during idle times of his computer) and the other being *application programmer* (the user who wants to run his own specific parallel iterative application on several volatile nodes). The resource provider will have a Daemon running on his host. The Daemon is the entity responsible for executing a Task and we consider it is busy and not available when a Task is executed on it (thereafter, we use the term Daemon and node indifferently). On the other side, the application programmer implements an application (using the Java language and the JaceV API) and actually runs it using the Spawner: this entity actually starts the application on several available Daemons.

Finally, the Dispatcher is the component in charge of 1) registering all the Daemons connected to the system and managing them (i.e. detect the eventual disconnections and replace the nodes) 2) distributing the Task objects of an application over the different available nodes, 3) detecting the global convergence of a running application, and 4) storing the backups of all the Tasks being executed.

Three-tier architectures are commonly used in fault tolerant platforms, like in Ninflet, XtremWeb, etc. However, JaceV has the advantage to enable both direct communications between computing nodes and multi-threaded programming, which is impossible with other existing environments. Furthermore, JaceV is the only one to implicitly provide an asynchronous iteration model by using primitives of its API. Therefore, JaceV is an original architecture.

3.3 The Dispatcher

The Dispatcher is the first entity to be launched for the environment. We consider it is running on a powerful and stable server. Therefore, all the data stored in this entity are considered as persistent. The Dispatcher is composed of three main components, 1) the *JaceVDispatchServer*, 2) the *GlobalRegister* and 3) the *ApplicationManager*.

The JaceVDispatcher is the RMI server that contains all the methods remotely invoked by the Daemons and the Spawner. It is launched when the Dispatcher starts and is continuously waiting for remote invocations.

The GlobalRegister registers all the Daemons connected to the JaceV system and also stores their current state (the 'alive' and the 'busy' states, this will be described in section 3.4).

Finally the ApplicationManager indexes all the *RunningApplication* objects of the system. A RunningApplication is a JaceV object that models an application being currently executed on the system: it contains for example attributes

[4] Java Virtual Machines

such as the URL where are available the corresponding class files of the application, the number of Tasks, the optional arguments, etc.

Each RunningApplication contains a single *Register* object, which is a subset of the GlobalRegister. During the execution, the Register is automatically updated in case of fault (due to a crash or a user disconnection) of one of the computation node (it models the whole configuration at time t of the nodes running a given application and the mapping of the Tasks over the Daemons).

The Dispatcher is also in charge of storing the Task objects saved (called *Backups*) during the computation in order to restart the application from a consistent global state in case of fault. A list stored in the RunningApplication object (the *BackupList*) indexes each Task composing an application. Hence, when a faulty node is replaced, the last Backup of the Task it was computing is sent to the new Daemon in order to restart computations. As iterations are desynchronized in the asynchronous model, the other nodes keep computing without stopping.

Finally, the RunningApplication object is responsible for detecting the global convergence and halting the application when convergence is reached. To do this, each RunningApplication object manages an array containing the local states of the nodes involved in the computation. This array is affected each time a local convergence message is received from the Daemons. When a node is in a local stable state (i.e. the relative error between the last two iterations on this node is greater than a given threshold) after a given number of iterations, it sends 1 to the Dispatcher, or else, it sends 0. The global state is computed on the Dispatcher by testing all the cells of the array, if they are all in stable state then the convergence is detected and the Daemons can stop computing.

To summarize the architecture of the Dispatcher, Figure 1 describes the main objects with the GlobalRegister on the left, the JaceVDispatcher (the RMI server) on the right and the ApplicationManager in the center.

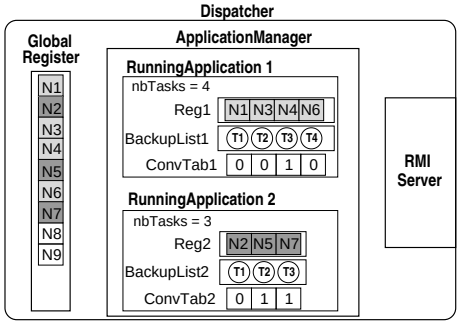


Fig. 1. Description of the Dispatcher elements.

In this example, nine Daemons are currently registered to the Dispatcher (nodes *N1* up to *N9* in the GlobalRegister). Only seven nodes are actually busy (i.e. computing an application). They appear in grey in the GlobalRegister (in the figure, we represent the nodes in different grey levels in order to differentiate

the application being executed on the corresponding Daemon). In the ApplicationManager, we can see that two applications are currently running, the first one (*RunningApplication1*) is distributed over four nodes (which are the nodes *N1, N3, N4* and *N6* in the corresponding Register called *Reg1*) and the second one (*RunningApplication2*) over three nodes (which are the nodes *N2, N5* and *N7* in the corresponding Register called *Reg2*). This figure also shows the Backups stored on the Dispatcher for each application being executed (*BackupList1* for the first application and *BackupList2* for the second one). Every BackupList contains a single Backup object for each Task running on a Daemon. The last elements appearing in the figure are the convergence arrays (*ConvTab1* for the first application and *ConvTab2* for the second one): with the values of *ConvTab1*, we can deduce that only Task *T3* (executed on node *N4*) is in a local convergence state for the first application. Concerning the second application, we can see that Tasks *T2* and *T3* (respectively running on nodes *N5* and *N7*) have locally converged to the solution.

3.4 The Daemon

When the Daemon is started, an RMI server is launched on it and is continuously waiting for remote invocations. Then, the Daemon 1) contacts the Dispatcher in order to obtain its remote RMI reference 2) remotely registers itself on the GlobalRegister of the Dispatcher (where this Daemon is then labeled as *available* because it has not been attributed an application yet), and 3) starts locally the *heartbeatThread*: this thread periodically invokes the *beating* remote method on the Dispatcher RMI server to signal its activity. The Dispatcher continuously monitors these calls to implement a timeout protocol: when a Daemon has not called for a sufficient long time, it is considered down in the GlobalRegister (i.e. it is labeled as *notAlive*). In case this node was executing an application, the Task initially running on it should be rescheduled to a new available Daemon by reloading the last Backup stored on the Dispatcher for the faulty node.

Once all those features are performed, the Daemon is initialized and ready to be invoked by the Spawner in order to actually run computation Tasks. The main objects composing the Daemon are mostly the same as in the Jace environment (interested readers can see [14] to have more details about the components of the Jace Daemon and their interaction). However, several objects have been deeply modified or added to the JaceV environment in order to ensure volatility tolerance. Those components are described in the following.

The Daemon contains the Register of the application it is running and this Register is automatically updated by the Dispatcher when faults occur during the execution. As the Register also contains the complete list of the nodes running a given application and the mapping of the Tasks over them, the Daemon is always aware of the topology of the system. This ensures direct communications are carried out between nodes because the Register contains the remote reference RMI for each Daemon. As a consequence, a given node can invoke remote methods on every Daemon running the same application. Furthermore, when a

node receives a new Register, the recipient of all the *Message* objects to be sent is automatically updated (if it has changed).

Concerning the Messages to send to other Daemons, as the asynchronism model is message loss tolerant, the Message is simply lost if the destination node is not reachable.

3.5 The Spawner

The Spawner is the entity that actually starts a user application. For this reason, when launching the Spawner, it is necessary to give some parameters to define this application: 1) the number of nodes required for the parallel execution, 2) the URL where the class files are available and finally 3) the optional arguments of the specific application.

Then, the Spawner sends this information to the Dispatcher that creates a new RunningApplication with the given parameters and a new Register composed of the required number of available nodes appearing in the GlobalRegister (which are then labeled as *notAvailable*). This Register is then attributed to the RunningApplication and sent to the Spawner.

Finally, when the Spawner receives the Register object, it broadcasts it to the whole nodes of the topology and then actually starts the computation on each of the Daemons.

The whole interaction between the JaceV entities is described in Figure 2. In this example, we can see the Daemon *N1* (fig.2(a)) and then a set of Daemons (*N2*, *N3* and *N4*, fig.2(b)) registering themselves to the Dispatcher. Those Daemons are then added to the GlobalRegister (*Reg*) and are labeled as *available* because no application has been spawned on the system yet.

In fig.2(c), the Spawner *S1* launches application *appli1* which requires two nodes. The Dispatcher creates then a RunningApplication object for this application and attributes it a Register object (*Reg1*) containing two available nodes of the GlobalRegister (*N1* and *N2* which are then labeled as *notAvailable* and appear in grey level in the GlobalRegister). The Register is sent to *N1* and *N2* (in order to permit direct communications between the two nodes) and the application is actually run by the Spawner on those two Daemons.

In fig.2(d), the Daemon *N2* crashes (or is disconnected by its user). However, as the asynchronous iteration model is used in JaceV, *N1* keeps computing and does not stop its job (the eventual messages to send to the Task running on *N2* will be lost until the node is replaced). The Dispatcher detects this disconnection and labels *N2* as *notAlive* in the GlobalRegister. *Reg1* is then updated in the RunningApplication object (*N2* is replaced by *N3* which is available) and this new Register is sent to the corresponding Daemons (*N1* and *N3*, the new one). Since then, *N1* is aware of the new topology of the system and updates the list of its neighbors (i.e it will no longer try to send messages to *N2* but will directly send them to *N3*). Finally, the Dispatcher sends the appropriate Backup to the new node of the topology and computations can restart on this Daemon.

After several minutes, the Daemon is launched again on node *N2* (fig.2(e)). It is then labeled as *alive* and *available* in the GlobalRegister.

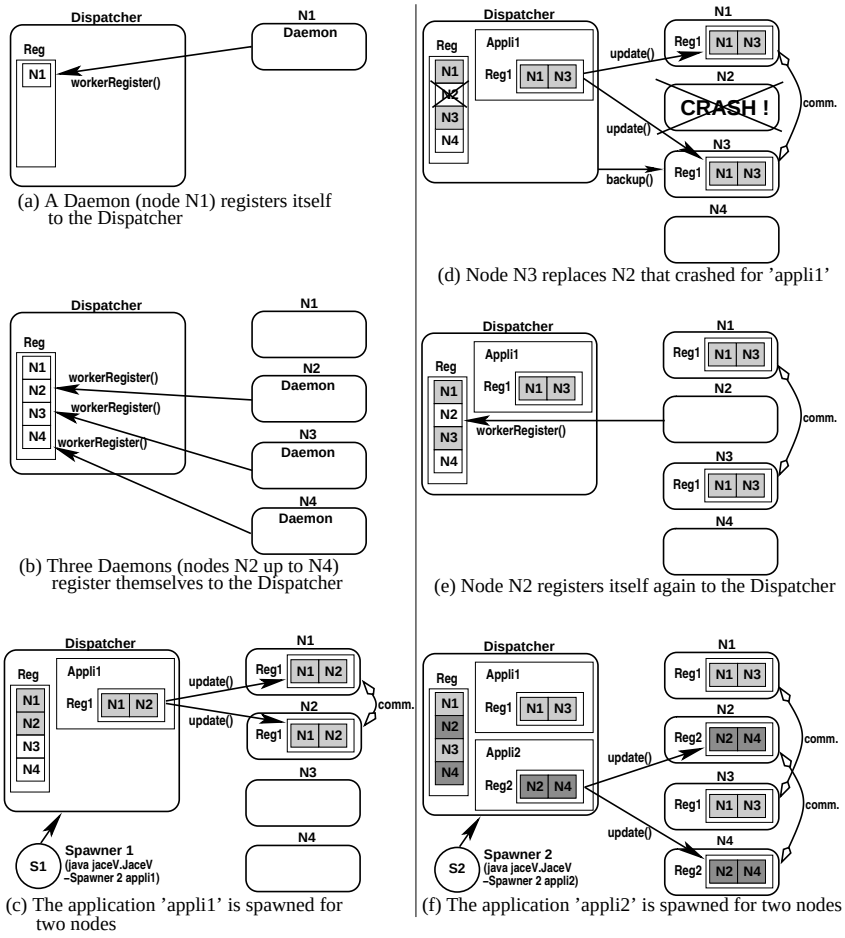


Fig. 2. The registering and spawning processes in JaceV.

Finally, in fig.2(f), the Spawner *S2* launches application *appli2* that requires two nodes. The Dispatcher creates the RunningApplication object for this application, attributes it a new Register (*Reg2*) which contains the last two available nodes of the GlobalRegister (*N2* and *N4*) and sends them *Reg2* in order to enable direct communication between these Daemons. At the end, *S2* actually starts computations on *N2* and *N4*.

4 Problem description

In this section, we describe the problem used for the experiments with JaceV. It consists of the Poisson equation discretized in two dimensions. This is a common problem in physics that models for instance heat problems. This linear elliptic

partial differential equations system is defined as

$$-\Delta u = f. \tag{1}$$

This equation is discretized using a finite difference scheme on a square domain using a uniform Cartesian grid consisting of grid points (x_i, y_i) where $x_i = i\Delta x$ and $y_j = j\Delta y$. Let $u_{i,j}$ represent an approximation to $u(x_i, y_i)$. In order to discretize (1) we replace the $x-$ and $y-$derivatives with centered finite differences, which gives

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2} = -f_{i,j} \tag{2}$$

Assuming that $\Delta x = \Delta y = h$ are discretized using the same discretization step h, (2) can be rewritten in

$$\frac{-4 * u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{h^2} = -f_{i,j}. \tag{3}$$

For this problem we have used Dirichlet boundary conditions.

So, (1) is solved by finding the solution of the following linear system of the type $A \times x = b$ where A is a 5-diagonal matrix and b represents the function f.

To solve this linear system we use a block-Jacobi method that allows us to decompose the matrix into block matrices and solve each block using an iterative method. In our experiments, we have chosen the sparse Conjugate Gradient algorithm. Besides, this method allows to use overlapping techniques that may dramatically reduce the number of iterations required to reach the convergence by letting some components to be computed by two processors.

From a practical point of view, if we consider a discretization grid of size $n \times n$, A is a matrix of size (n^2, n^2).

It should be noticed that, in the following, the number of components by processor is important and is a multiple of n, the number of components of a discretized line, and that the overlapped components is less important than this number of components. The solution of this problem using parallelism involves that each processor exchanges, at each Jacobi iteration, its first n components with its predecessor neighbor node and its last n ones with its successor neighbor node. The number of components exchanged with each neighbor is equal to n. In fact, we have only studied the case where the totality of overlapped components are not used by a neighbor processor, only the first or last n components are used because the other case entails more data exchanged without decreasing the number of iterations. So, whatever the size of the overlapped components, the exchanged data are constant.

Moreover we recall that the block-Jacobi method has the advantage to be solvable using the asynchronous iteration model if the spectral radius of the absolute value of the iteration matrix is less than 1, which is the case for this problem.

Finally, the Poisson problem implemented using the JaceV API has the skeleton described in Algorithm 1:

Algorithm 1 The Poisson problem skeleton using the JaceV API

```
Build the local Poisson submatrix
Initialize dependencies
repeat
  Solve local Block-Jacobi subsystem
  Asynchronous exchange of nonlocal data //with jaceSend() and jaceReceive()
  jaceLobalConvergence() //Local convergence detection
  jaceSave() //Primitive used to save the Task object on the Dispatcher
  jaceIteration++ //Increment the iteration number of the Backup to store
until jaceGlobalConvergence()
```

5 Experiments

For our experiments, we study the execution times of the application over 16 nodes according to n (with n varying from 500 up to 1800, which respectively corresponds to matrices of size 250,000×250,000 up to 3,240,000×3,240,000 because the problem size is n^2). An optimal overlapping value is used for each n. These experiments are performed with different configurations of processors and networks. For each configuration, we first run the application over 16 stable nodes, and then, for the execution in a volatile context, we launch 19 Daemons and run the application over 16 of them. In the last case, our strategy for volatility is to randomly disconnect each Daemon on average slightly less than two times during the whole execution of the application and reconnect it a few seconds later (i.e. there are approximatively about 30 disconnections/reconnections for each execution).

We choose to perform those series of tests with different configurations of processors and networks. According to processors, we use both homogeneous and heterogeneous processors. The first context consists of a 19-workstation cluster of Intel(R) Pentium(R) 4 CPU 3.00GHz processors with 1024MB of RAM. For the heterogeneous case, we use 19 workstations from Intel(R) Pentium(R) III CPU 1266MHz processors with 256MB of RAM up to Intel(R) Pentium(R) 4 CPU 3.00GHz with 1024MB of RAM. Then, we perform our tests with different network bandwidths.

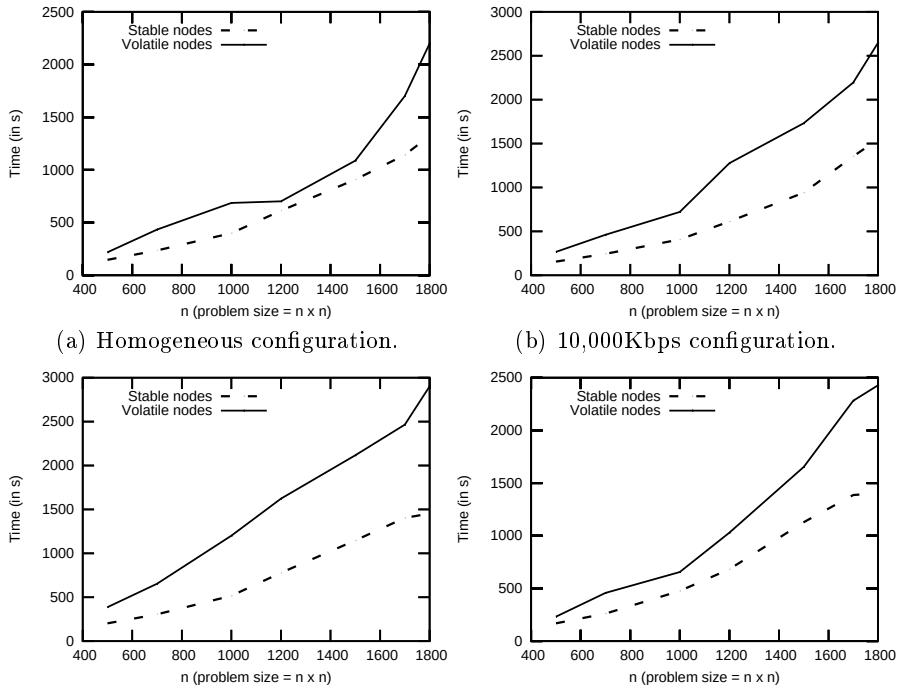
Finally our series of tests are performed using four configurations of processors and network, which are described as follows.

1. A configuration with homogeneous processors and an Ethernet 1Gbps network,
2. a configuration with homogeneous processors and a 10,000Kbps upload and download bandwidth,
3. a configuration with homogeneous processors and a 1,000Kbps upload and download bandwidth,
4. a configuration with heterogeneous processors and an Ethernet 100Mbps network.

For the second and the third configurations, each workstation of the cluster runs a Qos[5] script in order to limit the network bandwidth to 10,000Kbps (for configuration 2) and 1,000Kbps (for configuration 3).

Whatever the configuration used, the Dispatcher is running on an Intel(R) Pentium(R) 4 CPU 3.00GHz processor with 1024MB of RAM.

The results of the experiments are represented in figure 3 and each execution time is the average of a series of ten executions.



(a) Homogeneous configuration.

(b) 10,000Kbps configuration.

(c) 1,000Kbps configuration.

(d) Heterogeneous configuration.

Fig. 3. Execution times of volatile and non volatile contexts for the different configurations.

Analyzing the four figures, we deduce that JaceV supports rather well the volatile context. Indeed, although there are approximatively 30 disconnections during the whole execution, the ratio *volatile context execution time/stable context execution time* is always less than 2.5. Furthermore, at some point during the execution, less than 16 nodes are actually computing because more than 3 nodes are currently disconnected (they have not reconnected to the system yet). In this case, the alive nodes keep computing and are not waiting for the other Daemons to reconnect as it would occur in a synchronous execution.

[5] Quality of Service

We can also deduce that the lower the network bandwidth is, the greater the ratio according to the problem size is (this is particularly obvious in fig.3(c)). This is due to the fault detection and the restarting of the application. Indeed, when the Dispatcher detects the disconnection of a node (and eventually replaces it), it broadcasts the new Register object to all the alive nodes involved in the execution of the application. If the bandwidth is low, this action takes a certain time to be performed (because the size of the Register is not negligible). Hence, some Daemons would continue to send messages to the disappeared node during this period until the Register is actually updated on the Daemons. Furthermore, when the new Daemon replaces a faulty node, it must completely reload the Backup object from the Dispatcher. This object is rather important in terms of size, and it can take some time to deliver it on a low bandwidth network and to actually update it on the new Daemon. All those actions make the application much slower to converge to the solution.

Finally, comparing the execution times on homogeneous and heterogeneous workstations (respectively fig.3(a) and fig.3(d)) we can see that the curves are rather similar. As a consequence, we can deduce that JaceV does not seem to be that sensitive to the heterogeneity of processors for this typical application and perhaps for other similar coarse grained applications. This is undoubtedly due to the asynchronism which allows the fastest processors to perform more iterations.

6 Conclusion and Future Works

In this paper, we describe JaceV, a multi-threaded Java based library designed to build asynchronous parallel iterative applications and run them over volatile nodes. A goal of JaceV is to provide an environment with communications between computation nodes after each iteration, as it is necessary to run parallel iterative applications. JaceV uses the asynchronous iteration model in order to avoid synchronizations. Indeed, synchronous iterations would dramatically slow down the execution in a volatile context where nodes appear and disappear during computation.

The performance of the Poisson problem resolution show that JaceV is fully suitable for running asynchronous iterative applications with volatile nodes. We also remark that performances of JaceV are degraded if the network bandwidth gets very low. Experiments have been conducted with matrices of size 250,000×250,000 up to 3,240,000×3,240,000.

In future works, we plan to decentralize the architecture of JaceV in order to avoid bottlenecks on the Dispatcher. Some solutions to carry out those modifications lie in using for example a decentralized convergence detection algorithm, or storing Backups on computation nodes, and so, to reach a really P2P like environment.

References

1. Bertsekas, D., Tsitsiklis, J.: Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Englewood Cliffs NJ (1989)
2. SETI@home: http://setiathome.ssl.berkeley.edu
3. Baldeschwieler, J., Blumofe, R., Brewer, E.: Atlas: An infrastructure for global computing. 7th ACM SIGOPS European Workshop on System Support for Worldwide Application (1996)
4. Bahi, J., Miellou, J. -C., Rhofir, K.: Asynchronous multisplitting methods for nonlinear fixed point problems Numerical Algorithms, **15**(3, 4) (1997) 315–345
5. Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U., Takagi, H.: Ninflet: A Network based information Library for a global world-wide computing infrastructure. HPCN'97 (LNCS-1225) (1997) 491–502
6. Takagi, H., Matsuoka, S., Nakada, H., Sekiguchi, S., Sato, M., Nagashima, U.: a Migratable Parallel Object Framework using Java. In Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing (1998)
7. Aida, K., Nagashima, U., Nakada, H., Matsuoka, S., Takefusa, A.: Performance evaluation model for job scheduling in a global computing system. 7th IEEE International Symp on High Performance Distributed Computing. (1998) 352–353
8. Rosenberg A. L.: Guidelines for data-parallel cycle-stealing in networks of workstation. Journal of Parallel and Distributed Computing. **59** (1999) 31–53
9. Basney, J., Levy, M.: Deploying a High Throughput Computing Cluster. Volume 1, Chapter 5, Prentice Hall (1999)
10. Frommer, A. and Szyld, D.: On asynchronous iterations Journal of computational and applied mathematics. **23** (2000) 201–216
11. Bosilca, G., Bouteiller, A., Capello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. ACM/IEEE International Conference on SuperComputing, SC 2002, Baltimore, USA (2002)
12. Elnozahy, E.N., Alvisi, L., Wang, Y.M., and Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, **34**(3) (2002) 375–408
13. Bouteiller, A., Capello, Herault, T., Lemarinier, P., Magniette, F.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. ACM/IEEE International Conference on SuperComputing, SC 2003, Phoenix, USA (2003)
14. Bahi, J., Domas, S. and Mazouzi, K.: Combination of java and asynchronism for the grid: a comparative study based on a parallel power method. 6th International Workshop on Java for Parallel and Distributed Computing, JAVAPDC workshop of IPDPS 2004, IEEE computer society press (2004) 158a, 8 pages
15. Browne, J. C., Yalamanchi, M., Kane, K., Sankaralingam, K.: General Parallel Computations on Desktop Grid and P2P Systems. 7th Workshop on Languages, Compilers and Runtime Support for Scalable Systems. LCR 2004, Houston,Texas (2004)
16. Cappello, F., Djilali, S., Fedak, G., Hérault, T., Magniette, F., Néri, V. and Lodygensky, O.: Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. Future Generation Comp. Syst., **21**(3) (2005) 417–437