JaceP2P: an Environment for Asynchronous Computations on Peer-to-Peer Networks

Jacques M. Bahi   Raphaël Couturier   Philippe Vuillemin
Laboratoire d'Informatique de l'université de Franche-Comté (LIFC),
IUT de Belfort-Montbéliard, Rue Engel Gros, BP 27, 90016 Belfort, France
{jacques.bahi, raphael.couturier, philippe.vuillemin}@iut-bm.univ-fcomte.fr

Abstract

Using Peer-to-Peer (P2P) networks is a way to federate a large amount of processors in order to solve large scale scientific problems. Those networks are decentralized, highly dynamic and composed of heterogeneous machines. The goals of our work is to compute large scale scientific iterative applications on P2P networks. We propose JaceP2P, a multi-threaded Java based library designed to build asynchronous parallel iterative applications. Using this library, it is possible to run such applications on a set of dynamic and heterogeneous machines organized in a decentralized and P2P fashion.

1 Introduction

Large scientific problems are complex systems to solve and usually require in order to be executed the simultaneous use of several computation resources. Scientific applications can be categorized in two kinds: direct algorithms and iterative ones. The former class consists in finding the problem solution in one step. The latter proceeds by successive approximations of the solution and the class of problems that can be solved using this kind of algorithms is large. It features, for example, sparse linear systems, ordinary differential equations [1, 2], partial differential equations [3, 4], Newton based algorithms, polynomial root finders [5]... In this paper, we are interested in iterative algorithms. In the parallel execution of those algorithms, communications must be performed between computation nodes after each iteration in order to satisfy all the computing dependencies.

As supercomputers and clusters with a large amount of nodes are expensive, scientists are not always able to run their applications with enough computational resources. "Desktop Grid Computing" environments (also called "Global Computing" environments) can deliver massive computing power to run scientific applications using cycle stealing concepts (i.e. the nodes contribute to the computation of an application during their idle times). However, those environments cannot be used to run iterative applications as long as communication is restricted to the master-slave model of parallelism (i.e. communications between slaves are impossible). As a consequence, Peer-to-Peer computing seems to be an elegant solution to run parallel iterative applications on a large amount of nodes.

The term "Peer-to-Peer" (P2P) refers to a class of systems and applications that uses distributed resources (called "peers") to perform a function in a decentralized and autonomous manner. Conceptually, P2P computing is an alternative to the centralized and client-server models of computing in which there is typically a single server and many clients executing independent tasks. P2P systems assume that the computing environment is highly dynamic. As a consequence, a fault tolerance strategy must be designed to detect disconnections, and also to re-issue computation jobs to other participants.

Concerning iterative applications, the most interesting strategy to face the problem of dynamicity seems to be the use of the asynchronous iteration model. Indeed, as idle times and synchronizations are suppressed in this model (because processors never wait for computing dependencies before starting an iteration), we do believe the asynchronous solution is the most suitable one in a P2P network.

Asynchronous algorithms can be used in a significant set of problems. Indeed, scientific applications are often described by systems of differential equations which leads, 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 multisplitting converges asynchronously (see [6–8] and the references therein).

In our work, we do not consider the synchronous iteration model because it does not seem to be convenient neither for the heterogeneity and scalability nor for the dynamicity. Indeed, due to the synchronizations in this case, all the nodes involved in the computation of an application would stop computing when a single disconnection occurs.

In this paper, we describe JaceP2P, a multi-threaded Java based library designed to build asynchronous parallel iterative applications and execute them on P2P networks of PCs (used during their idle times).

In the following section, we recall the main characteristics of P2P systems and the different architectures it is possible to build. Section 3 presents several existing P2P platforms (general environments and distributed computing ones). In Section 4 we describe the JaceP2P system and its different entities. Section 5 presents the functionalities of the platform. Section 6 presents the scientific iterative problem we implemented and ran using JaceP2P (the Poison Problem). In Section 7, we describe the experiments we led to evaluate the performance of the JaceP2P environment. In Section 8, we conclude the paper and some perspectives are given.

2 Overview of P2P networks

This section introduces the characteristics of P2P networks and the topology usually built.

2.1 Characteristics

Decentralization This characteristic makes the implementation of P2P platforms difficult to build in practice because there is no centralized server with a global view of all the peers in the network. Those platforms should avoid or at least reduce centralization to the maximum.

Scalability P2P platforms usually manage and use a huge amount of nodes. Scalability is often limited by the amount of centralized operations (e.g. synchronization and coordination) that need to be performed. As a result, it is necessary to develop algorithms that reduce or avoid centralizations and synchronizations within the system.

Dynamicity/Fault resilience As the most frequent environment is personal home computers connected to the Internet, peer groups frequently change (e.g. due to network link failures or because a user turns off his computer). As a consequence, it is necessary to design a strategy to detect connections and disconnections of peers.

Anonymity/Privacy A user might not want anyone or any service provider to know about his involvement in the system. Furthermore, as P2P environments can gather several thousands of resources, login identification mechanisms are sometimes impossible in such contexts.

Portability P2P architectures share heterogeneous resources (in term of hardware, operating system, network bandwidth, etc.). As a consequence, the model should enable codes to have similar portability. This is a necessary prerequisite for coping with heterogeneous configurations.

2.2 Topology

P2P networks are usually divided in three categories of architectures: 1) centralized ones, 2) decentralized ones (also "Pure P2P" architectures) and 3) hybrid ones.

Centralized architectures Centralized topologies are very close to Client/Server ones. In this model, a stable central server indexes all the peers of the system and stores the information about their content. Upon request from a peer, the central index selects another peer in its directory that matches the request. Thereafter, communications are performed directly between the two peers. However, centralization may generate bottlenecks and can present some scalability limits (it requires for example a bigger server and a larger bandwidth when the number of peers and requests increases).

Decentralized architectures In "Pure P2P" architectures, each peer has exactly the same capabilities and can both act as a client or a server at the same time. In this model, there are no advertisement of shared resources, each request from a peer is "flooded" (broadcasted) to directly connected peers, which themselves flood their neighbors. This action is performed until the request is answered or a maximum number of flooding steps is reached. This solution is tolerant to any failure or disconnection because of the decentralization. However, it is more difficult to carry out in the Internet context because a peer should either forward requests on the whole network to join the system or use a host list with known IP addresses of other peers currently registered in the system.

Hybrid architectures The third topology shares the advantages of the two former classes: it is a decentralized architecture where several nodes (called "Super-Peers") have the capability to index and manage a set of peers connected to the system. A Super-Peer is linked to other ones so that it can forward requests from a peer to another one not directly connected to it. As a consequence, all the peers of the system are connected to a set of Super-Peers, which avoids overheads and bottlenecks due to centralizations.

3 Related work

Project JXTA [10] is an open-source project initiated to standardize a set of protocols for building any kind of P2P application (e.g. file sharing, distributed computing...). JXTA manages peers in a totally decentralized fashion (using the Pure P2P model) and supports fault resilience through the redundancy of components and the elimination of a single point of failure. However, due to its low-level platform nature, much of the task management and the support for different computing models is left to the developer of the application. As a consequence, JXTA provides a lot of P2P functionalities but is rather complicated to use for the development and the execution of complex computing applications.

JNGI [11, 12] aims to provide a general framework for the performing of P2P distributed computing on grids of JXTA peers. Like JXTA, this environment supports node dynamicity through data redundancy. JNGI defines 1) a Monitor peer group that manages all the activity of the system, 2) a Worker peer group which is responsible for the execution of the tasks, and 3) a Task Dispatcher peer group that distributes the tasks on the workers. However, this framework does not enable communications between Worker peers executing different tasks. As a consequence, the environment is only suitable for applications with independent tasks.

ProActive [13] is a grid Java library for parallel, distributed, and concurrent computing. It provides a comprehensive API[1] in order to simplify the programming of applications that are distributed on Local Area Network (LAN), on cluster of workstations or on Internet Grids. However, although this environment provides direct communication between peers, it is not possible to specify a particular task to send data to (as is required when dealing with iterative applications) since messages are always broadcasted to the neighbors of a peer.

Jace [18] (Java Asynchronous Computation Environment) 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[2]. However, this environment does not support dynamicity and disconnections of nodes. As a consequence, it is not suitable to run applications on P2P networks.

[1] Application Programming Interface

[2] Remote Method Invocation

4 Presentation of JaceP2P

4.1 The goals of JaceP2P

Jace is fully suitable for running parallel iterative asynchronous applications in a Grid computing context (i.e. where nodes do not disappear during computations). However, programming applications on top of P2P networks requires properties beyond that of a simple sequential programming or even parallel and distributed programming. In [19], we present JaceV (JaceV stands for Jace Volatile), a fully centralized volatility tolerant platform to compute iterative algorithms on volatile nodes.

In this paper, we propose JaceP2P, the P2P and decentralized version of JaceV. It is a programming and execution environment designed to implement iterative asynchronous applications. This platform is build on top of JaceV but is especially developed to run this kind of applications in a P2P context (i.e. a dynamic set of volatile and heterogeneous peers organized in a decentralized fashion). The platform allows direct asynchronous communications between peers in order to exchange local results during the iterations (i.e. the computing dependencies). Furthermore, JaceP2P is a multi-threaded environment. This functionality enables the overlapping of communications by computations which is necessary when dealing with the asynchronous iteration model. Consequently communications costs are hidden. According to the problem and the topology, they may vary.

JaceP2P is developed using the Java language. Executions are usually slower using Java than using other programming languages. However, this ensures the portability of the platform on different heterogeneous resources (i.e. PCs with different operating systems and hardware), as it is always the case on P2P networks. Furthermore, as communications are performed using Java RMI, a node of the system directly communicates with the other ones using their *RMI stub* objects (their remote RMI references which contain all their location data without employing login information). As a consequence, the JaceP2P system does not require user login nor NFS access information and only uses IP addresses when peers join the network (this is described in Section 5.1). After entering the JaceP2P network, all the peers communicate using only the RMI stub objects.

In JaceP2P, fault-tolerance and dynamicity is managed by checkpoint-based rollback recovery techniques (for a classification of these techniques, interested readers can see [20]). In this context, a *consistent global* *checkpoint* is a set of local checkpoints, one for each process, forming a consistent global state in order to enable the system to restart properly after a node failure. Any consistent global checkpoint can be used to restart process execution upon failure. As JaceP2P is based on the asynchronous iteration model, a consistent global state is always achieved whatever the set of local checkpoints taken into consideration (i.e. it is not necessarily the set of local checkpoints at a given iteration like in the synchronous iteration model). As a consequence, the alive processors do not need to stop computations when a failure occurs, only the replacing one should restart job from the last available checkpoint stored for this task.

4.2 Architecture of the system

The JaceP2P environment is based on the hybrid P2P topology model (see Subsection 2.2) and is composed of three different types of entities which are JVMs[3] communicating with each other:

- the *Daemons* which are the computing peers in charge of running an application in a parallel fashion (thereafter, we use the term *Daemon* and *computing peer* indifferently),
- the *Super-Peers* which are responsible for registering the Daemons entering the system,
- and the *Spawners* which are the peers that actually start the execution of applications on a set of available Daemons registered to the JaceP2P network. The Spawner is the only entity of the system to be stable. In future work, we plan to study how to make it tolerant to failures.

In order to ensure the fault tolerance of the JaceP2P system, the Daemons are also responsible for storing some checkpoints (called the *Backup* objects in the environment) of a set of its neighbors executing the same application (this is described in subsection 5.4).

A JaceP2P application is a set of *Task* objects running on Daemons and exchanging messages and data in order to solve a given iterative problem. Several applications can be executed in the JaceP2P network at the same time, but a Daemon can only run a single Task at a given time. Indeed, we consider a Daemon is busy and is not available to run another application when a Task is already executed on it. Furthermore, for performance considerations, only a single Daemon can be launched on a given computer.

The user of the JaceP2P system could play two types of roles, one being 1) a *resource provider* and the other being 2) an *application programmer* (the user who wants to run his own specific parallel iterative application). On the one hand, the resource provider will have a Daemon running on his host and executing a Task object during idle times of the computer. On the other hand, the application programmer will implement an iterative problem and, by using the Spawner, will actually launch the application on several available Daemons of the JaceP2P network. A user application is a SPMD[4] Java program which uses JaceP2P methods by extending the Task class of the environment.

[3]Java Virtual Machines

[4]Single Program Multiple Data

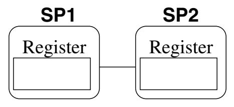
5 Functionalities of the JaceP2P environment

5.1 Joining the JaceP2P Network: the Bootstrapping

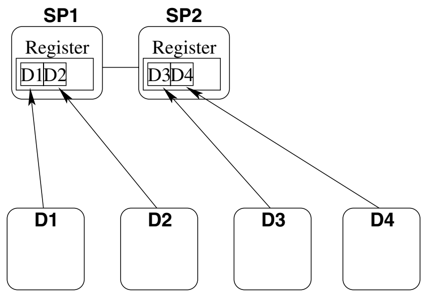
The Super-Peers represent the entry points of the JaceP2P network. They are linked together and they are in charge of indexing the Daemons entering the system. Each Super-Peer manages a *Register* object containing all the RMI stubs of the Daemons directly connected to it. This is described in Figure 1 in which we give the example of a two Super-Peer network. This configuration example is used in the following figures of the current section.

Figure 1(a) represents the two Super-Peers (called $SP1$ and $SP2$ linked to one another) which are currently waiting for Daemons connections. As no Daemon has registered yet, each *Register* is empty.

When a Daemon is started on a peer, an RMI server is launched on it and is continuously waiting for remote invocations. The corresponding RMI stub has to be sent to one of the Super-Peers of the system in order to register to the JaceP2P network. This action of joining a P2P network is usually called the *bootstrapping* and can be difficult to carry out because of the dynamicity of the system. Our strategy consists in storing on each Daemon a list of IP addresses of several Super-Peers supposed to be frequently available and waiting for computing peers to connect. A Daemon randomly chooses one of the Super-Peers in the list to register to it. If the Super-Peer is not reachable, we consider it is down and the computing peer has to choose another IP address in the list. This action is repeated until a Super-Peer is reachable. Once the Daemon has located the JaceP2P network, it can actually send its RMI stub in order to register to the system. As a consequence, only the bootstrapping in JaceP2P requires the use of IP addresses. Thereafter, only RMI stubs are used to locate the different entities of the network.



(a) A network of two Super-Peers are waiting for Daemon connections



(b) Four Daemons register to the JaceP2P system, each one by sending its RMI stub to one of the Super-Peers

Figure 1. A network of Super-Peers and the registration of Daemons

This is described in Figure 1(b) in which four Daemons connect to the system by locating one of the Super-Peers and by sending their RMI stubs: Daemons *D1* and *D2* register to Super-Peer *SP1* and Daemons *D3* and *D4* register to *SP2*. Each Super-Peer stores in its Register the RMI stubs of the Daemons connected to it. Thereafter, the communications between those entities are performed through the stubs and no longer using IP addresses.

Once the Daemons are registered, they are available to run any application launched on the JaceP2P system.

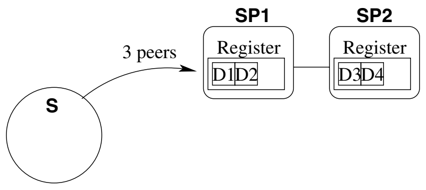
5.2 Launching an application

The Spawner is the JaceP2P entity run by a user (the application programmer) in order to actually launch an application on the system. When running the Spawner on his computer, the user must specify the following parameters:

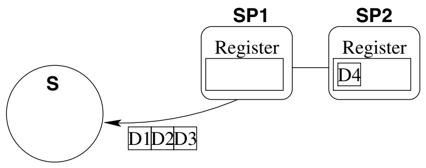
1. the URL of a web server where the corresponding class files of the application are available,
2. the number of required computing nodes,
3. the optional arguments of the application.

Once it has been launched, the Spawner connects to the Super-Peer network (using the bootstrapping strategy described in the previous subsection) and should reserve Daemons in order to run the application with the required number of computing nodes.

If the Super-Peer located has not registered a sufficient number of Daemons to directly answer a Spawner request, it reserves computing nodes registered to other Super-Peers of the network. This is described in Figure 2(a): a Spawner *S* asks *SP1* for three peers to run an application. As *SP1* has only registered two peers (*D1* and *D2*), it reserves the third one (*D3*) on *SP2* and sends the list of RMI stubs back to the Spawner (as shown in Figure 2(b)). Thereafter, the corresponding Daemons are no longer registered to the Super-Peers (i.e. the stubs have been removed from the Register) but directly to the Spawner.



(a) A Spawner asks the Super-Peers for three peers in order to run an application



(b) The Super-Peers reserve three Daemons and send the corresponding RMI stubs to the Spawner

Figure 2. The reservation of computing peers

When receiving the list of RMI Stubs, the Spawner stores it in its own Register object (as described in Figure 3 in which this Register is called *AppliReg*). In the following, we call this object the *Application Register* because it models, at time *t*, the whole configuration of the peers running a given application and the mapping of the Tasks over the Daemons.

This object is sent to each Daemon of the list (the peers *D1, D2* and *D3* in the figure) with the different parameters specified above. Then, computations can start on each computing peer by locally creating an instance of the Task object (which actually represents the user application and is available through the byte-code located at the URL specified above).

As the Application Register is then stored on each Daemon, all the RMI stubs of its neighbors are available. As a consequence, a given computing peer can invoke remote methods on every Daemon running the same application. This enables direct communications between peers in order to exchange computing dependencies during the iterations.

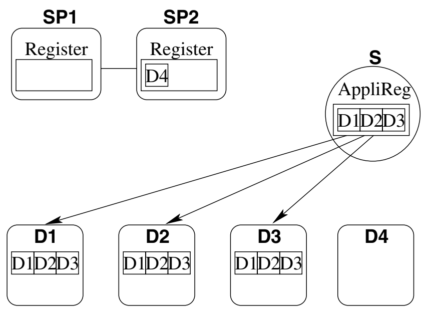


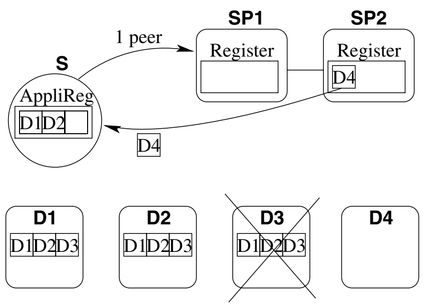
Figure 3. Broadcast of the Application Register to the Daemons.

5.3 Detecting peer disconnections

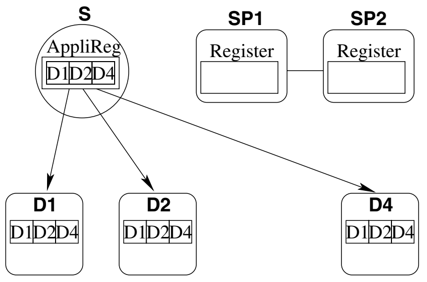
Daemons can get disconnected of the JaceP2P network either 1) when they are not involved in computations yet or 2) when they are currently running an application. In the first case, a computing peer is linked to the Super-Peer it has registered and periodically signals its activity to it using a *heartbeat* mechanism. The Super-Peer continuously monitors these calls to implement a timeout protocol. When a Daemon has not called for a sufficient long time, it is considered down and is removed from the Super-Peer Register. In the same way, if a Daemon detects a Super-Peer failure (i.e. when it is no longer reachable by the computing peer to signal its activity), the Super-Peer is considered down and all the Daemons connected to it should locate another Super-Peer in order to register to it.

In the second case, Daemons running a Task object are no longer registered to one of the Super-Peers but directly to a Spawner. This Spawner is then responsible to detect computing peer failures using the same heartbeat strategy as described above (i.e. when they are running an application, the Daemons do no longer signal their activity to a Super-Peer but to a Spawner). When the Spawner detects Daemon disconnections, the corresponding RMI stubs are removed from the Application Register. The Spawner must then contact the Super-Peer network in order 1) to reserve available Daemons and 2) to update the Application Register. This is described in Figure 4(a): the Spawner S detects the disconnection of *D3* and removes the corresponding RMI stubs from *AppliReg*. As the asynchronous model is message loss tolerant, any message to be sent by other peers to *D3* is lost, and the alive nodes keep computing their tasks.

Then, *S* asks *SP1* for one available peer. As *SP1* has no registered Daemons, it forwards the request to its neighbor (*SP2*) which has registered *D4*. The RMI stub for *D4* is then sent to the Spawner *S* which can update its Application Register (called *AppliReg* in the figure).



(a) The Spawner asks the Super-Peers for one peer in order to replace the faulting one



(b) The Spawner broadcasts the Application Register to the new set of peers

Figure 4. Replacing a disconnected computing peer

This updated list of RMI stubs is sent to the new set of computing peers so they can locate all their neighbors. It is described in Figure 4(b) in which *AppliReg* is sent to *D1* and *D2* (the peers already involved in the computation) and also to *D4*, the new computing peer of the topology.

When a peer receives a new list of neighbors, the recipient of all the messages to be sent is automatically updated (if it has changed). Furthermore, the message is simply lost if the destination peer is not reachable (i.e. if it has failed again or has not been replaced yet).

5.4 Checkpointing and restarting tasks upon failure

In the JaceP2P environment, the Daemon is the entity in charge of storing a set of Task objects saved by other peers during their computation in order to restart the application from a consistent global state in case of failure. Those objects are called *Backups* and they actually model the local checkpoints of the application. Hence, a task initially running on a peer that failed should be rescheduled to a new available Daemon by reloading the last Backup stored on one of its neighbors. According to the considered scientific problem, it can be interesting to checkpoint tasks at each given number of iterations (and not at each iteration). Hence, the application programmer specifies the frequency of the checkpointing process with the *JaceSave()* method of the JaceP2P API.

In practice, a list stored on the Daemon indexes a set of checkpoints (one for each task the current peer must backup). During the whole execution of an application, a peer always saves its current Task object on the same set of neighbors (in a round-robin fashion). We call "backup-peers" the nodes responsible for storing the different versions of a given Task object (i.e. the local checkpoints for a task at different iteration numbers).

This is described in Figure 5 with the example of a four-task execution (four Daemons, *D1* up to *D4*, are executing four Tasks, respectively *T1* up to *T4*).

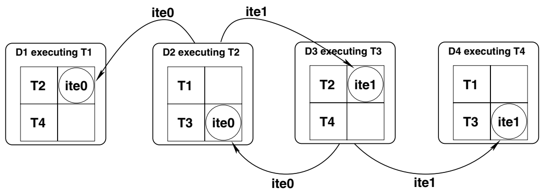


Figure 5. The checkpointing process performed for the Daemons running tasks T2 and T3.

In this example, we consider the backup-peers are the left and right neighbors of each Daemon. We also consider the checkpointing process is performed at each iteration and is represented on the figure for the tasks $T2$ and $T3$ at the first two iterations ($ite0$ and $ite1$). As an example, during the executions, the Backups of Task *T2* are always stored on the same set of peers which are 1) the Daemon running the Tasks *T1* (the Backups stored for even numbers of iteration) and 2) the Daemon running *T3* (the Backups stored for odd numbers of iteration). Thereafter, the Backup stored at iteration *ite2* for task *T2* would then replace that of iteration *ite0* on $D1$

When a disconnected peer is replaced, the new Daemon asks the backup-peers for the iteration number of each of its local checkpoints remotely stored. Then, it reloads the last Backup (i.e. the checkpoint of higher iteration number) in order to restart computations.

This is described in Figure 6: we consider the Daemon running $T3$ (the peer $D3$) fails and is replaced by $D5$. $T3$ should then be rescheduled to the new peer. $D5$ searches for the different versions of the Backup objects available for $T3$. A Backup at iteration 6 is available on $D2$ and another one is available on $D4$ for iteration 7. The version at iteration 7 being the most recent Backup of $T3$ available in the system, this task can restart on $D5$ from this local checkpoint.

As iterations are desynchronized in the asynchronous model, the other peers keep computing without stop-

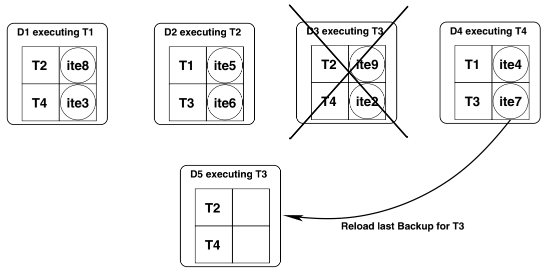


Figure 6. Reloading the most recent available Backup object on a new Daemon.

ping. By using this strategy, it is convenient to choose a sufficient number of backup-peers in order to ensure that at least one Backup is available for a given Task object if several of those peers have failed. If not, computations for this task should restart from the beginning (i.e. at iteration 0).

5.5 Convergence detection

The Spawner is the JaceP2P entity responsible for detecting the global convergence and halting the application when convergence is reached. In the case of iterative algorithms, the convergence is commonly associated with the relative error between the last two iterations. As long as the error is greater than a given threshold, the system is seen as unstable and when this error becomes lower, it is assumed that the convergence is reached.

In the case of parallel iterative algorithms, the convergence detection is directly linked to the knowledge of the global state of the system since global convergence is achieved only when each peer is in a stable state. In order to obtain this global knowledge, we centralize the local convergence state of all peers over the Spawner: this entity manages an array containing the states of all the peers involved in the computation of the corresponding application. This array is updated each time a local convergence message is received from the Daemons. When a peer is in a local stable state during a given number of iterations, it sends 1 to the Spawner. If its local state changes from a stable state to an unstable state, it sends 0.

The global state is computed on the Spawner by testing all the cells of the array. If they are all in stable state, the convergence is detected and the Daemons can stop computing.

The JaceP2P system can tolerate disconnections of Super-Peers and Daemons. The only entity which should be stable is the Spawner because it manages the Application Register and the global convergence array.

However, as the Spawner is running on the computer of the application programmer in order to execute his own problem, it is assumed that he should not turn off his machine until the execution has finished.

6 Problem description

In this section, we describe the problem used for the experiments with JaceP2P. 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

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

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. So this problem entails a sparse matrix representation.

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 $n^2 \times n^2$.

In the following, it should be noticed that the number of components by processor is large and is a multiple of n (n being the number of components of a discretized line). It is also assumed that the number of overlapped components is less important than n. The solution of this problem using parallelism involves that at each Jacobi iteration, each processor exchanges 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 let us remind the reader that the block-Jacobi method has the advantage of being 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.

7 Experiments

For our experiments, we consider a P2P network of heterogeneous and dynamic machines connected to the JaceP2P environment. The network is actually composed of:

- a set of three Super-Peers running on Intel Pentium 4 CPU 2.40GHz processors with 512MB of RAM,
- a set of about 100 Daemons launched on workstations from Intel Pentium III CPU 1266MHz processors with 256MB of RAM up to Intel Pentium 4 CPU 3.00GHz with 1024MB of RAM,
- a Spawner running on an Intel Pentium 4 CPU 2.40GHz processor with 512MB of RAM.

In this configuration, some machines are connected to an Ethernet 1Gbps network and others to an Ethernet 100Mbps network. Hence, both machines and networks are heterogeneous. For our tests, the checkpointing process is performed every five iterations and 20 backup-peers are used to store the checkpoints of each task.

In the experiments, we study the execution times of the Poisson problem according to n in different dynamic contexts of the P2P network. The application is launched over 80 peers of the JaceP2P network described above and we compare the total execution times of the application with different numbers of disconnections (actually from 0 up to 50 disconnections). The peers are randomly disconnected during the execution, and they are reconnected about 20 seconds later. Although only 80 peers are used to execute the application, this work is the only one, to the best of our knowledge, to run a real numerical problem with interdependent tasks on a P2P network of heterogeneous and dynamic machines.

In the tests, n varies from 2000 up to 5000, which respectively corresponds to matrices of size 4,000,000×4,000,000 up to 25,000,000×25,000,000 because the problem size is n^2. An optimal overlapping value is used for each n. The results of the experiments are represented in Fig. 7 and each execution time is the average of a series of ten executions.

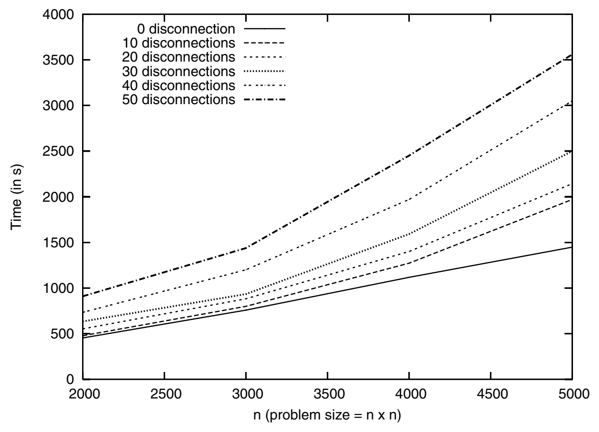


Figure 7. Execution times of the Poisson problem (launched with 80 processors) according to n with different numbers of disconnections

In the following, the ratio:

$$\frac{computing\ time\ per\ iteration}{communication\ time\ per\ iteration} \tag{4}$$

allows us to understand the behavior of the conducted experiments. In our tests, the number of processors running the application is fixed and the size of the global problem varies. As a consequence, the local vector size for each processor becomes greater as n increases. When the local vector size is small, a processor must perform more iterations to reach a local convergence state because the ratio (4) is small. In opposition, when the local vector size is large, this ratio is greater and the number of iterations to reach the local convergence state is smaller. Indeed, in the asynchronous iteration model, a processor may not receive computing dependencies from all its neighbors at a given iteration. In the case no dependency is received at all during a given iteration, the next one will not make the computation progress toward the solution of the problem, since no update has been received. This particular case is more frequent when the ratio (4) is small (i.e. when the problem size is small).

This phenomenon is actually the case in our experiments: concerning the executions without disconnection, the problem for $n = 2000$ (i.e. a problem of size 4,000,000) needs on average about 100 iterations to reach the global convergence, whereas for $n = 5000$, about 40 iterations are necessary. This obviously shows that the number of iterations without update is more important with a small problem than with a larger one.

As a consequence, disconnections have not the same impact with small or large problem sizes. Concerning small problems (i.e. small ratio (4)), failures of nodes are less significant on the execution times (i.e. they disrupt less computations) than concerning larger ones. Indeed, with a small ratio, the number of useless iterations is more important.

Despite this fact, figure 7 shows that JaceP2P supports disconnections rather well: for $n = 2000$, we compare the execution times between the case without disconnection and the case with 50 disconnections (which is the case with the higher number of disconnection for our experiments). We remark that, at the maximum, the failures slow down the execution times with a factor 2. For $n = 5000$, the same phenomenon is visible with a maximum factor of 2.5, which is not so different from the previous case (i.e. for $n = 2000$). We can then deduce that although there are a large amount of disconnections, this factor does not increase much.

Consequently, JaceP2P is an efficient architecture to design and run iterative asynchronous algorithms on P2P networks with different problem sizes (so with different values of ratio (4)).

8 Conclusion and Future Works

In this paper, we describe JaceP2P, a peer-to-peer programming and execution environment. This platform allows to implement real scientific iterative applications with computation dependencies between tasks. It also permits to efficiently run those algorithms using dynamic and heterogeneous computers organized in a decentralized fashion. As a consequence, disconnections of computing nodes during the execution and heterogeneity of processors (and bandwidth) do not disturb the final results of a JaceP2P application.

The platform uses the asynchronous iteration model in order to avoid synchronizations between interdependent tasks. Indeed, synchronous iterations would dramatically slow down the execution in a dynamic and heterogeneous P2P network. As a consequence, if a numerical algorithm converges asynchronously on stable nodes, the use of JaceP2P on dynamic peers ensures that this algorithm would also converge on a P2P network of heterogeneous machines, even if disconnections occured. To the best of our knowledge JaceP2P is the only platform dedicated to implementing asynchronous iterative application on P2P networks.

Experiments have been carried out with a linear problem (the Poisson problem). Using JaceP2P, the execution times are not dramatically slowed down when an important number of disconnections occur during computations.

In future work, we plan to implement other scientific iterative applications using the JaceP2P environment. The class of problems that can be implemented with this platform is large and features, for example, nonlinear applications, nonstationary PDE systems, etc. Furthermore, it would be interesting to study the behavior of those algorithms in a very large scale P2P network composed of several hundreds of peers. Some aspects of JaceP2P probably needing to be improved (broadcast of register, convergence detection, ...).

Acknowledgment

This work was supported by the "Conseil Régional de Franche-Comté".

References

[1] E. Hairer and G. Wanner. *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, volume 14 of *Springer series in computational mathematics*, pages 5–8. Springer-Verlag, Berlin, 1991.

[2] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press Inc., New York, 1995.

[3] G. D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.

[4] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, 1998.

[5] D. A. Bini. Numerical computation of polynomial zeros by means of alberth's method. *Numerical Algorithms*, 13:179–200, 1996.

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

[7] J. M. Bahi, J. C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3, 4):315–345, 1997.

[8] A. Frommer and D. Szyld. On asynchronous iterations. *Journal of computational and applied mathematics*, 23:201–216, 2000.

[9] Project Napster, 2001. http://www.napster.com/.

[10] Project JXTA, 2003. http://www.jxta.org/.

[11] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *GRID*, pages 1–12, 2002.

[12] J. Verbeke and N. Nadgir. JNGI: P2P Distributed Computing. Technical report, Sun Microsystems, Inc, Palo Alto, California, September 2003.

[13] ProActive. INRIA, 1999. http://www-sop.inria.fr/oasis/ProActive.

[14] D. Caromel, C. Delbe, and A. di Costanzo. Peer-to-peer and fault-tolerance: Towards deployment based technical services. In *Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, Paris, France, January 2006.

[15] S. Djilali. P2P-RPC: Programming scientific applications on peer-to-peer systems with remote procedure call. In *CCGRID*, pages 406–413, 2003.

[16] F. Cappello, S. Djilali, G. Fedak, T. Hérault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Comp. Syst.*, 21(3):417–437, 2005.

[17] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP. *Lecture Notes in Computer Science*, 2104:130–??, 2001.

[18] J. M. Bahi, S. Domas, and K. Mazouzi. Combination of java and asynchronism for the grid : a comparative study based on a parallel power method. In *18th IEEE and ACM Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 158a, 8 pages, Santa Fe, USA, April 2004. IEEE computer society press.

[19] J. M. Bahi, R. Couturier, and P. Vuillemin. JaceV: a programming and execution environment for asynchronous iterative computations on volatile nodes. In *VECPAR'06 (to appear)*, Rio de Janeiro, Brazil, july 2006.

[20] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.