A Fault Tolerant and Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Financial Option Pricing

Sébastien Bezinne, Virginie Galtier, Stéphane Vialle, Françoise Baude, Mireille Bossy, Viet Dung, Ludovic Henrio

A Fault Tolerant and Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing in Finance.

Sébastien Bezzine, Virginie Galtier, Stéphane Vialle
Supélec, 2 rue Edouard Belin, 57070 Metz, France
bezzine@metz.supelec.fr, *Virginie.Galtier@supelec.fr*, *Stephane.Vialle@supelec.fr*

Françoise Baude, Mireille Bossy, Viet Dung Doan, Ludovic Henrio,
INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93 - 06902 Sophia Antipolis Cedex, France
First.Last@inria.fr

Abstract

This paper introduces a Grid software architecture offering fault tolerance, dynamic and aggressive load balancing and two complementary parallel programming paradigms. Experiments with financial applications on a real multi-site Grid assess this solution.

This architecture has been designed to run industrial and financial applications, that are frequently time constrained and CPU consuming, feature both tightly and loosely coupled parallelism requiring generic programming paradigm, and adopt client-server business architecture.

1. Introduction and Project Overview

The presented work aims at designing and implementing a Grid architecture and some Grid applications for financial risk analysis. This research takes place within the 'ANR GCPMF' project from academic laboratories in computer science and mathematics, banks, and IT companies.

Financial institutions (banks, insurance companies, major industries...) are linked by numerous and interwoven deals, but first of all, they are market participants with sophisticated needs in terms of exchange which are met by financial derivative instruments like options. Recently, derivative dealers have promoted derivatives instruments to hedge or customize market-risk exposure. For this reason, derivative instruments are sometimes called "risk management products". Moreover, new requirements introduced by Basel–II standard to face market risk (but also counterpart risk, credit risk, clearing risk), lead financial institutions to compute daily risk measurements of their portfolios.

Thus, financial risk analysis includes different kinds of computations. Computations can be short or long (sometimes called *day* and *night* computations), and embarrassingly parallel or requiring communications. But usually, all of them have to respect time constraints. For example, results must be known to close each deal on time, or before the next day to adjust investments function of the market evolution. The computational power provided by Grid computing appears as an interesting solution to support financial risk analysis, but the Grid needs to be adapted to business applications and business actor usages. In this context we consider multi-site but *private*, access restricted Grids.

The Grid solution architecture introduced in this article is named PicsouGrid[1], and its main goals are:

- to create a Grid of distributed computing *services*, started at boot-time and remaining *up*, ready to quickly process any client request,
- to achieve fault detection and fault tolerance, in order to fulfill requests, even in non-reliable environments,
- to bound failure recovery overhead so that, with a security margin, applications can meet their time constraints,
- to provide a comfortable programming environment for parallel programs both with or without inter-task communications.

This work also consists in making experiments on real large distributed architectures.

This article is organized as follows. Section 2 introduces some related works. Sections 3 and 4 introduce two different Grid programming environments and paradigms; both are available in PicsouGrid and allow to easily implement parallel algorithms both with or without inter-task communications. Section 5 describes the PicsouGrid complete architecture. Section 6 introduces the main features of fi-

[1] *Picsou* is French for Scrooge McDuck, Walt Disney character

nancial risk analysis, and presents in detail the pricing of the most demanding out of three European option types. Section 7 introduces the first experimental performances of PicsouGrid. Lastly, Section 8 summarizes the main results achieved and the next steps of this project.

2. Related Works

Lots of existing frameworks for distributed computing feature a master-worker architecture, either in a global computing or peer-to-peer environment [4, 3] or in a Grid context [5]. As an example, OurGrid [5] is a complete solution for running Bag-of-Tasks applications on computational grids. Platform and Datasynapse propose industrial grid environments for financial computations. They are mainly used by banks to run Bag-of-Tasks applications on computational grids. Among all the research being handled for master-worker architectures, it is worth noticing recent efforts in order to adapt a component view of such systems [8]. In general, master-worker architectures support computation for bags of independent tasks similar to the application described in this paper; however, implementing an architecture above ProActive and JavaSpace middlewares will allow us to support interactions between tasks, and thus ease the programming of more complex pricing. Moreover, as underlined in [1] a hierarchy of masters as presented here seems particularly adapted to Grid infrastructures to feature load-balancing and fault-tolerance.

Our objective is to design a fault-tolerance protocol based on application-level checkpointing. To our knowledge the only work implementing such a challenging approach is [12]. Adapting this approach to PicsouGrid requires designing a new fault-tolerance mechanism, probably inspired from [7] (which achieves a purely middleware fault-tolerance for ProActive).

3. ProActive Environment

In a high-performance computing context, asynchronous and collective communications should be accessible to programmers, so the usage of RMI (*Remote Method Invocation*) is not sufficient. Consequently, we have developed *ProActive*: it is a platform that features 1) a programming model for distribution, 2) mechanisms that enable it to be used as a high-level portable and flexible middleware for running applications on virtually any sort of computing infrastructure (desktop machines, clusters, grids...).

ProActive is available as a 100% Java library, for parallel, distributed, concurrent computing with security of communications and mobility of the activities [6]. RMI is used as the serialization and default transport layer, or alternatively HTTP. ProActive uses an *active object* (AO) programming model. AOs are remotely accessible via method

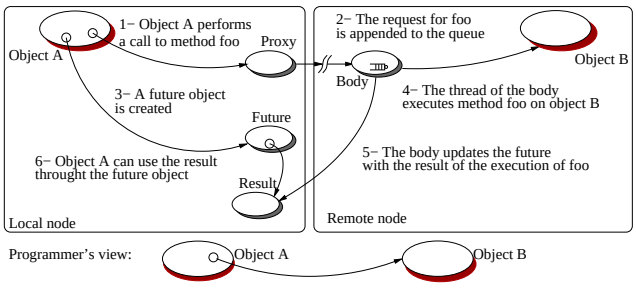

Figure 1. ProActive remote method call

invocation, automatically stored in a queue of pending requests (Figure 1). Each AO has its own thread of control and is granted the ability to decide in which order incoming method calls are served (FIFO by default). Method calls on AO are asynchronous with automatic synchronization: automatic *future objects* as a result of remote methods calls, and synchronization are handled by a mechanism known as *wait-by-necessity* [9]. Using an original checkpointing protocol [7] makes it possible to restart the whole application from a previous global state in case of failure. Active objects can also wrap native code, including SPMD MPI based parallel codes. From the base model of point-to-point *ProActive* communications, an extension towards groups of active objects is available.

ProActive also provides a *Descriptor based Deployment Model*, which alleviates the code from references to: machine names, submission protocols, registry/lookup protocols, heterogeneous file-transfer protocols. This allows the deployment of applications on sites using heterogeneous protocols, without changing the application code. Integration of heterogeneous file transfer with resource acquisition protocols allows on-the-fly deployment, i.e. deployment of the Grid application together with the Grid middleware.

4. JavaSpace Service

JavaSpace [11] is a Jini service enabling programs to exchange objects through a virtual shared memory. It is an evolution from a work done nearly two decades ago at Yale University with the Linda system but benefiting from Jini and Java object-oriented paradigm and portability. Several commercial implementations proposing various associated tools and demonstrating different performances exist.

The JavaSpace API proposes three main methods: *write* to put an object into the JavaSpace, and *read* and *take* to retrieve a copy of an object or the object itself from the JavaSpace. These methods operate according to two modes: blocking or non blocking. A lease is associated with each entry and objects are removed from the space when the lease expires. Object retrieval is done by matching a template (*associate lookup*): a program might indicate the class of

the desired object, and optionally the value for some of the object's attributes. Additional functionalities include a notification triggered by write operations and a set of batch operations. This API is both simple yet rich enough to enable fast and easy development of a large range of distributed applications, in particular those with high communication requirements. Moreover, Jini and JavaSpaces services offer three fault-tolerance mechanisms, which might prove useful for distributed systems like Grids prone to partial failures (see section 5.4).

JavaSpace constitutes a comfortable and highly portable middleware for programming fault-tolerant distributed applications using a shared memory paradigm. But it is a common statement that software virtual shared memories usually fail to scale. Yet the authors could not find serious work verifying those hypotheses by running and measuring a real application using JavaSpaces on a large-scale grid. The introduction of JavaSpace in this project thus serves two purposes: (1) to offer a partially fault-tolerant shared memory in Java, and (2) to evaluate the limits of JavaSpace for applications on large-scale grids.

5. PicsouGrid Architecture

5.1. Architecture Principles and Overview

A previous project [10] and preliminary discussions with banking institutions have stressed the necessity for the distributed part of the architecture to transparently coexist with usual industrial architectures. As a result, we designed our PicsouGrid architecture so that the server acts as an entry point to the system distributed over a cluster or grid. To foster extensibility, a hierarchical structure was adopted: the server controls a set of sub-servers which in turn control a large number of workers. Figure 2 depicts this architecture.

The different processes composing this architecture need to be deployed and started beforehand on the different processors. Then PicsouGrid can process requests on demand.

5.2. Grid Programming Methodology

Multiple programming paradigms: Financial algorithms fall into different categories, each of which is best adapted to a particular programming paradigm (shared memory, message passing, remote procedure call). Large and loosely coupled computations will be distributed on a large number of *worker* nodes, potentially managed by several *SubServer* nodes, using ProActive programming environment. To ease the programming of parallel algorithms requiring inter-task communications, we plan to investigate and compare two solutions: the first simply consists in relying on ProActive communication to handle transparently any inter-*worker* and *worker*-to-*SubServer* communications; the second relies on JavaSpace to handle some of these communications. Such a JavaSpace can be instantiated on demand and is suitable at least when a *SubServer* and all its *workers* are deployed on a same cluster with fast communication network (see experiments in section 7).

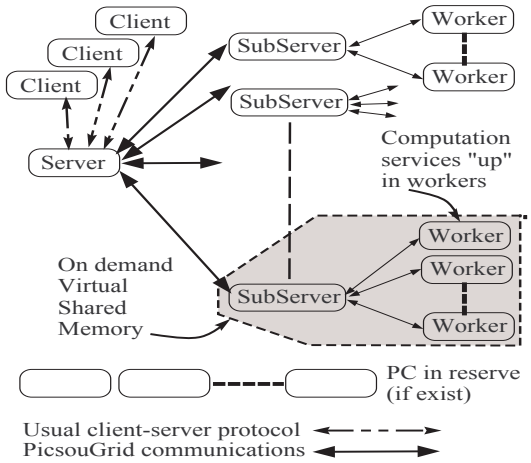

Figure 2. Global PicsouGrid architecture

PicsouGrid has been designed and implemented to include both ProActive and JavaSpace paradigms, in order to be a generic Grid architecture supporting the implementation of various parallel algorithms and different programming strategies.

Skeletal programming using Java *generics*: Overall, the PicsouGrid architecture has been designed in such a way that the programmer of the financial application only provides some sequential codes for the core of *workers* and to gather results in *SubServers* and *Servers*. Figure 3 shows the main class diagram of the PicsouGrid software architecture. The *user code* is encapsulated within the three bottom classes, that inherit from three PicsouGrid classes (*Server*, *SubServer* and *Worker*). These PicsouGrid classes take in charge the Grid initialization, the on-demand shared memory management, the fault tolerance achievement, and the dialog set up with client applications. The *Server* and *SubServer* classes need to create and manipulate objects of user defined classes (*UserSubServer* and *UserWorker* on Figure 3). These Java classes are *generics* parametrized with the user-defined classes, so the user-defined classes can have any name and PicsouGrid can be specialized to each application.

This approach is known as *skeletal programming*, where skeletons provide an overall architecture, further personalized according to each specific application [2]. Finally, the programmer of a financial application does not need to care about deployment or fault-recovery. As long as he/she extends our classes, PicsouGrid itself deals with those issues.

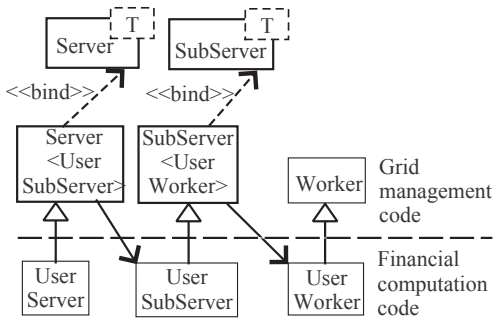

Figure 3. PicsouGrid **software architecture**

5.3. Main Load Balancing Mechanisms

Traditional dynamic load balancing: Resources of a large PicsouGrid can be heterogeneous and unreliable. A dynamic load balancing is mandatory to achieve good performances in any case. We split each financial computation into a large set of elementary tasks, that are dynamically distributed on the *workers*: each *worker* receives an elementary task to process and, asks for a new one when it has finished. This classical strategy has been implemented both with ProActive and with JavaSpaces:

– In the case of a JavaSpace, some initial data and tasks are put in the virtual shared memory by a *SubServer*. Each *worker* retrieves a task when the computation starts or when it has finished its previous task, and puts its results in the shared memory to be collected by the *SubServer* or read by other *workers*. Sometimes the *SubServer* or some *workers* provide new tasks to be processed in the JavaSpace.

– Implementing dynamic load balancing with ProActive in PicsouGrid requires to write a little bit more code in the *SubServer*. The *SubServer* manages a *group* of *workers* and sends a first task to each *worker*. Then it waits for the result of any *worker*, stores and analyzes this result and sends a new task to the *worker*.

These mechanisms can hierarchically be extended to the distribution of a set of tasks from the *Server* to the *SubServer*, thus ensuring dynamic load balancing of parallel computations on a large number of processors. It can be a pure ProActive, a pure JavaSpace or a mixed hierarchy.

Aggressive task distribution: Many risk analysis are based on Monte Carlo simulations, and need to run at least a fixed number of simulations (Q) to achieve the required accuracy. To achieve these stochastic computations as fast as possible, considering heterogeneousness of the machines and possible failures, a solution consists in sending tasks to workers until the *SubServer* has collected enough results (not just until having sent enough tasks to the *workers*). Then, useless tasks still running on *workers* are canceled by the *SubServer*. This strategy takes advantage of the fastest machines in the Grid, but generates more communications that may slow down the Grid. Another solution consists in sending big tasks to the P *workers* of the *SubServer*: like $Q/(P-1)$ simulations per task. On a large number of processors, without failure the execution is just a little bit longer. It remains unchanged when one failure appears, and increases only when several processors fail. This solution generates the minimum amount of communications, but slows down the Grid when the processors are heterogeneous.

5.4. Fault Tolerance Strategy and Implementation

Applicative fault tolerance: To detect faults in a simple way, the server regularly pings its *SubServers* and a *SubServer* regularly pings its *workers*. If the probed element fails to respond in time, it is considered as faulty. To minimize the amount of results lost by a failure, *SubServers* regularly checkpoint the received results. When a worker disappears, its *SubServer* first requests a node from the reserve pool. Next, it restarts a *worker* on that node and sends it the task the faulty *worker* was in charge of. If the reserve pool is empty, the system runs with less *workers*. A slightly more complex situation arises when a *SubServer* fails to respond. In that case, the *Server* requests a new node from the reserved pool; if the pool is empty, the *Server's* node is chosen since it does not perform a lot of computations. A new *SubServer* is started and the *Server* sends it the interrupted task, and the last check-pointing state from the dead *SubServer*; each *worker* from the initial group is re-attached to the new *SubServer*. The global computing time depends on the failure timing and on the number of concomitant failures (see section 7).

Middleware fault tolerance: Both ProActive and JavaSpaces provide fault tolerance mechanisms at the middleware level. ProActive can regularly checkpoint (serialize) Active Objects (AO) and handle communication events accordingly [7]. When an AO disappears ProActive re-creates it, inserts it automatically in the application and re-starts all the AOs from the last checkpoint. This middleware mechanism is an efficient solution to replace AOs with complex communication scheme, like *workers* with inter-*worker* communications. But it can be slow due to the regular serialization of an entire AO.

JavaSpace and Jini services offer 3 mechanisms to ease the development of fault-tolerant applications. (1) The shared memory can be parameterized to be saved onto the disk. (2) The JavaSpace service can be started as an activatable one to be re-activated upon failure. (3) Using a Jini transaction service, distributed transactions can be

used when writing or taking objects to guarantee the ACID[2] properties traditionally offered by database transactions. Those mechanisms introduce an overhead, which partially depends on the location of the services and network and system properties.

Mixed and collaborative fault tolerance: One of our main future objectives is to establish a collaboration between middleware and application fault tolerance mechanisms. ProActive methods of checkpointing and AO restoration can be overloaded at application level to optimize the frequency of checkpointing and the amount of data saved. JavaSpace accesses do not need to be systematically associated to a transaction; delicate operations can be identified at application level and protected at middleware level through transactions. Such mixed fault tolerance strategies should decrease checkpointing and fault recovery overheads.

6. Financial Risk Analysis

In the present work, we focus on options, one of the main instruments of financial risk management [15]. The term *option* usually refers to a contract giving to its owner the right but not the obligation to purchase (*call option*) or to sell (*put option*) a specified amount of transferable interest depending on the *underlying* asset, within a specified time span. Call and put options are characterized by an exercise price (or *strike*) K, which is the trading price of the underlying fixed by the contract. The option vanishes after a maturity date T. Options contracts are also characterized by their time exercise rule. In this work, we experiment on the case of the options which can be exercised only at the maturity date. These options are called European options and are widespread on all the market places.

Let us detail the payoff of a call/put option, that is the amount of money performed by the option: at the maturity date T, the holder of the call compares the market price S_T of the underlying with the strike K. If $S_T > K$, the holder exercises the call. In that case, the benefit for the holder is $S_T - K$ (and the loss for the seller is $K - S_T$). If $S_T \leq K$ the holder does not exercise. One resolves the two situations by saying that the payoff of the European call is $\max(0, S_T - K)$. Similarly, it is not difficult to see that the payoff of the put option is $\max(0, K - S_T)$. A call is an insurance against the increase over K of the market value underlying at a future date. The holder must pay a premium, corresponding to the risk transfer of the holder position on the seller of the option.

[2]ACID: Atomicity, Consistency, Isolation, Durability. Properties required to maintain a coherent state.

Option pricing: The fundamental principle of the option pricing (more generally of derivatives valuation) is *the arbitrage pricing under risk-neutrality*. If two assets have the same payoffs, they must have the same market price: the arbitrage price. The risk neutrality is the position of an agent that expects the same return from the risky assets S_t and a risk-free bond B_t for the same initial investment. For instance, the bond value could be $B_t = \exp(rt)$, where r is the risk-free continuous interest rate. On a given market (S_t, B_t), for a given risk-neutral probability $\mathbb{P}^*$ on this market, the option pricing theory [16, 17] states that the arbitrage price at time 0 of an option is the expected value, under $\mathbb{P}^*$, of its discounted payoff. Hence, at time 0, the premium P_0 of a put option, with strike K and maturity T is

$$P_0 = \mathbb{E}^* \left[\exp(-rT) \max(K - S_T, 0) \right]. \tag{1}$$

The computation of the mathematical expectation above implies to introduce a stochastic model for the dynamic of S_t starting from the observed price S_0. The most popular stochastic model for the underlying is the (multidimensional) Black-Scholes model, described by the stochastic differential equation (SDE)

$$S_t = S_0 + \int_0^t r S_s ds + \int_0^t \sigma S_s dW_s \tag{2}$$

where S_t is the vector of prices at time t of the underlying, σ is the volatility vector of the underlying assets and $(W_t, t \geq 0)$ is a correlated Brownian motion of dimension the number of assets. For multidimensional underlying assets or complex payoffs like look-back or barrier options, there is no explicit formula for the premium and numerical simulations are required. In this work, we experiment on Monte Carlo methods which can be applied for various payoffs and which are particularly well adapted for large dimensional underlying assets. For example, the underlying of the basket option reproducing call/put on the CAC 40 Euronext index is of dimension $d = 40$.

In order to approximate the expectation in (1), the Monte Carlo method consists in computing the arithmetic mean of $nbMC$ independent pseudo random simulations of the option payoff $\max(0, K - S_{T,i})$, according to the stochastic model (2)

$$\begin{aligned} &\mathbb{E}^* \left[\exp(-rT) \max(0, K - S_T,) \right] \\ &\simeq \frac{1}{nbMC} \sum_{i=1}^{nbMC} \left[\exp(-rT) \max(0, K - S_{T,i}) \right]. \end{aligned}$$

The computation of each $S_{T,i}$ implies to discretize in time the SDE (2): we compute $S_{T,i}$ using an iterative procedure $S_{(k+1)\Delta t,i} = F(S_{k\Delta t,i}, G_{k,i})$, for a given $S_{0,i}$, (see for instance, (3) below). The $(G_{k,i}, 1 \leq i \leq nbMC, 1 \leq k \leq$

/* Before the simulations: */
Compute L by the Cholesky decomposition.
/* On receipt of a task, each worker does: */
for $i = 0$ **to** $task\_size - 1$ **do**
  for $t = 0$ **to** $timeSteps - 1$ **do**
    Generate Z of random vector of law $N(0, \mathbb{I})$.
    Calculate the vector $G = L * Z$.
    for $j = 0$ **to** $d - 1$ **do**
      $S_j = S_j * \exp((r - \sigma_j^2/2)\Delta t + \sigma_j * G_j * \sqrt{\Delta t})$
    end for
  end for
  /* Compute the basket price P_i ****/***
  $P_i = \sum_{1 \leq j \leq d} p_j * S_j$
  /* Compute the payoff */
  $X_i = max(K - P_i, 0)$
  /* Compute the mean on the task */
  $task\_meanPut = \sum_{1 \leq i \leq task\_size} X_i$
  $task\_varPut = \sum_{1 \leq i \leq task\_size} (X_i)^2$
end for
/* Each worker return $task\_meanPut$ ****/***
/* and $task\_varPut$ ***to the Server */***
/* The Server does: */
$meanPut = \frac{\exp(-rT)}{nbMC} \infty_{1 \leq k \leq nb\_task} task\_meanPut_{(k)}$
$varPut = \frac{\exp(-2rT)}{nbMC} \infty_{1 \leq k \leq nb\_task} task\_varPut_{(k)} - (meanPut)^2$
/* Confidence interval at 95% of the put premium: */
Lower interval = $meanPut - 1.96 * \frac{\sqrt{varPut}}{\sqrt{nbMC}}$
Higher interval = $meanPut + 1.96 * \frac{\sqrt{varPut}}{\sqrt{nbMC}}$

Figure 4. Pricing European basket put option

$T/\Delta t)$ is a family of independent random variables of dimension $d \geq 1$ [14].

Interestingly, the number of simulations grows according to the statistical precision sought, proportionally to $nbMC$. There is a clear evidence that the grid computing power could serve in achieving the required precision, within a short delay. For each of the $nbMC$ simulations, the dimensions of the problem are the time scale of the maturity date (days, months, years), the time step (from one hour to one day), the dimension d of the underlying assets.

Monte Carlo simulations need pseudo-random generation with good independency properties to well converge, so random generators can not simply be distributed over a set of workers to be solved in parallel [13]. This issue has not yet been addressed in this project. A distributed RNG leads to better reliability.

Experimented distributed algorithms: We developed three distributed pricing algorithms for the general European option, the barrier option and the basket option. We describe the architectural and algorithmic principles, in the case of an European option on a basket of d assets, the other cases having very similar architecture. When d is large ($d = 40$ in ours experiments), basket option is particularly time consuming among European option types. The underlying prices is a vector $(S_t^j, j = 1, \ldots d)$. We consider a basket put option of strike K: for a fixed vector $(p_j, j = 1, \ldots, d)$ of the weight of each stock in the basket, the basket option payoff is $\max(0, K - \sum_{j=1}^{d} p_j S_T^j)$ and the corresponding approximated premium is:

$$\frac{1}{nbMC} \sum_{i=1}^{nbMC} \left[\exp(-rT) \max(0, \sum_{j=1}^{d} p_j S_{T,i}^j) \right].$$

We fix the form of the correlation matrix Cor of the Brownian motion (W_t): for a fixed correlation parameter $0 \leq \rho << 1$, $Cor_{i,i}$ is set to 1 and $Cor_{i,j}$ is set to ρ. We highly notice that one stock price of the basket can not be simulated independently from the others because of the correlation between them. The time simulation procedure is:

$$S_{(k+1)\Delta t,i}^j = S_{k\Delta t,i}^j \exp((r - \frac{\sigma_j^2}{2})\Delta t + \sigma^j G_{k,i}^j \sqrt{\Delta t}),$$
$$\text{for } 1 \leq j \leq d,\ 1 \leq k \leq \frac{T}{\Delta t},\ 1 \leq i \leq nbMC, \qquad (3)$$

where, the random vector $(G_{k,i}^j, j = 1, \ldots, d)$ could be easily sampled by noting that $G = LZ$. Z is a vector of d independent random variables of law $N(0, 1)$ (easy to simulate) and L is the lower triangular matrix, computed by the Cholesky decomposition applied to the matrix Cor.

We present a distributed algorithm for pricing the basket option on Figure 4. One basket trajectory simulation consists in the simulation of the d prices S_t^j. Because of the correlation in the model, the trajectory simulation of the prices must be synchronized. For this reason, we distribute only the number of (independent) basket simulations. A task sent to a worker consists in asking it to simulate a number $pack\_size$ of simulations. The main problem regarding load balancing will be to fix the optimal value of $pack\_size$ according to the total number of simulations $nbMC$ and the dynamic load of the processors of the grid.

7. Experimental Performances

Performances on sound Grid: We have evaluated PicsouGrid with a one year maturity European put option on a basket of 40 correlated stocks computed on the basis of 10^6 Monte Carlo trajectories (multidimensional Black-Sholes model with one step a day, 25% volatility per stock, 0.5% correlation, the mean precision is about 10^{-3}). Experiments were run on Grid'5000, the French experimental Grid (9 PC clusters across France), without introducing failure.

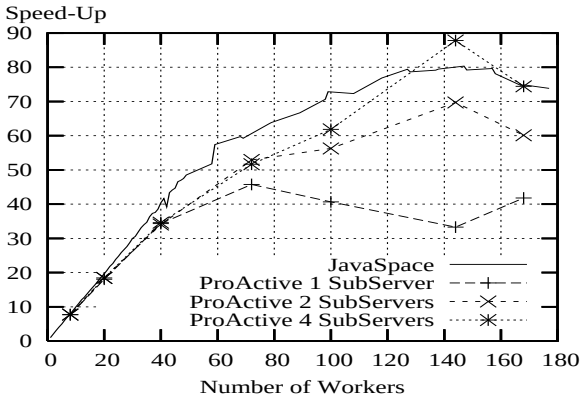

Figure 5. Speed up on a 1-site Grid (cluster)

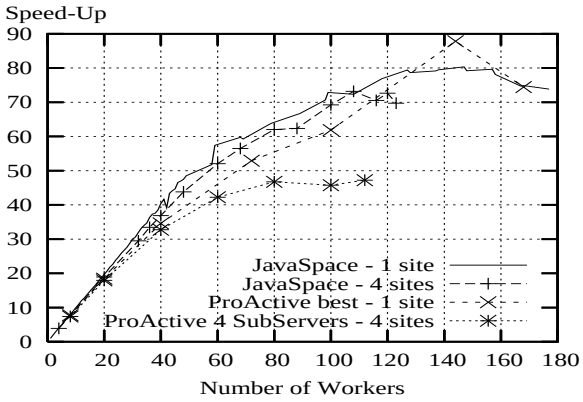

Figure 6. Speed up on a 4-site Grid

Using ProActive the number of *SubServers* need to be adapted to the size of the Grid, see Figure 5. In our experiment on one site, one *SubServers* is enough to manage up to 40 *workers*, then 2 *SubServers* are better to manage around 70 processors, and then 4 *SubServers* are desirable to manage more processors. Identifying the best configuration is strategic and need to be investigated in the future. Without using fault tolerant mechanisms, SUN JavaSpace performances are better up to approximately 130 processors, and decrease beyond. Future experiments will investigate the use of a hierarchy of JavaSpaces associated with fault tolerance mechanisms.

We have deployed PicsouGrid on 4 sites using Proactive, with 1 *SubServer* and up to 23 *workers* per site (Figure 6). Compared to the best configurations on one site a significant slow down appears for 80 processors, where the speedup slows down from 55 to 47. Using JavaSpace without any fault tolerance mechanisms, the slow down is small up to 112 processors, but amounts to 8.5 for 120 processors. Several sites offers more easily a large amount of processors, but communications take longer and limit the speedup, and it appeared difficult to dispose of a sound multi-site Grid. Many technical and unexpected problems have limited our experiments and further motivate the need of fault tolerance mechanisms.

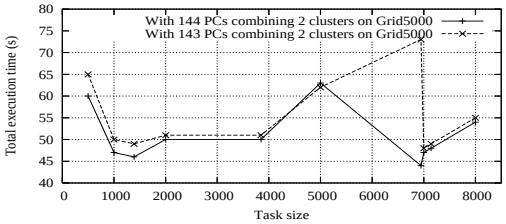

Figure 7. Impact of the task granularity

Failures and *aggressive* load balancing: When distributing independent Monte Carlo simulations, the heterogeneity and volatility of Grid machines can be addressed by the *aggressive task distribution* introduced in section 5.3, and defining an adequate trade-off for the task size. Figure 7 shows the execution times of a basket pricing, function of the task size on a nominal Grid of 144 processors and on a faulty Grid of 143 processors. No reserve processor is considered, the fault-resilience is entirely supported by the dynamic load balancing and the *aggressive task distribution* strategies.

Small task size is adapted for highly volatile and heterogeneous Grids. On Figure 7 the failure impact is very limited for small task size, but their numerous communications slow down the Grid. Optimal task size on sound Grid (nb-simul/nb-proc = 6945 on Figure 7) leads to minimal execution times, but to double time on our faulty Grid because all processors run one task and one processor has to run a second task to achieve the required amount of simulations. Little bit greater task size is adapted to homogeneous and lightly volatile Grids, like 7000 on Figure 7: more simulations than required are computed on the sound Grid with limited overhead, and the required amount is still achieved in the same time on the Grid with one faulty processor.

Aggressive task distribution achieves good performances on sound and faulty Grids when computing independent Monte Carlo simulations. Future work is to adjust the task size automatically.

Failures and fault recovery overhead: To test our fault tolerance mechanisms at applicative layer using ProActive, we have repeated a basket pricing computation on a 32 PC cluster at Supelec and deliberately caused some node failures. The testbed configuration is a basket pricing computation based on 10^7 simulations, split into 100 tasks of 10^5 simulations and run on 18 *workers* controlled by 3 *SubServers* and 1 *Server*. 22 PCs of the cluster were used, and 10 PCs remained available to replace faulty ones (see section 5.4).

Simultaneous Failures	$[t_{min} - t_{max}]$	$\bar{t}$	$\Delta\bar{t}$
no failure	[52.0s - 55.0s]	53.7s	-
1 *worker*	[52.4s - 57.3s]	54.1s	+0.4s
2 *workers*	[55.9s - 56.6s]	56.1s	+2.4s
4 *workers*	[60.1s - 61.8s]	61.0s	+7.3s
6 *workers*	[67.6s - 69.5s]	68.7s	+15.0s
1 *SubServer*	[63.0s - 63.8s]	63.3s	+9.6s
2 *SubServers*	[65.1s - 65.9s]	65.5s	+11.8s
3 *SubServers*	[67.3s - 69.4s]	68.0s	+14.3s

Table 1. Fault tolerance overheads

Two PicsouGrid mechanisms participate to limit failure impact: the fault tolerance mechanism tries to use PCs from the reserve pool to replace faulty ones, and the dynamic and aggressive load balancing mechanism supplies more work to other *workers*. When a failure happens on a *worker* at the beginning of a task computation, the time computation lost is low; when it happens at the end, just before the result is returned to the *SubServer*, the lost of time is higher. We have considered only this worst case when we have experimented simultaneous failures of 1 to 6 *workers*, and of 1 to 3 *SubServers* (see Table 1). It is worth noticing the PicsouGrid architecture supports large failures recovery without stopping. Moreover, the overhead (Δt) is less than 1s to replace a faulty *worker* and less than 10s to replace a faulty *SubServer*. To not replace faulty processors would lead to greater delays for current and next executions of the basket pricing service.

8. Conclusion and Perspectives

This article has presented a fault tolerant and multi-paradigm Grid software architecture, experimented for financial computations on a multi-site Grid. It has been designed to support both large bag-of-tasks problems and parallel algorithms requiring communications, offering simultaneously remote method invocation of active objects and on demand memory sharing. We achieved good performances with these two paradigms for European option pricing, using up to 190 processors on a cluster and up to 120 processors on a 4-site Grid. Fault-tolerance and load balancing are realized transparently for the programmer, based on processor replacement and dynamic and *aggressive* load balancing. These strategies have been successfully tested on a 32 PC cluster. We experienced technical difficulties with large scale experiments on a multi-site Grid because Grid'5000 innovative experimental environment still suffers from deployment and reservation weaknesses.

Future Works:
- to experiment with more tightly-coupled applications, such as American pricing,
- to improve fault-tolerance thanks to a collaboration between application-level and generic middleware-level fault-tolerance mechanisms,
- to improve scalability by 1) providing a better job pre-provisioning for workers, and 2) providing better strategy to re-balance load after several sub-server failures.

Acknowledgment: This research is supported by the French "ANR-CIGC GCPMF" project and by the "Region Lorraine", and Grid'5000 has been funded by ACI-GRID.

References

[1] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *CCGRID*, 00, 2003.
[2] M. Alt and S. Gorlatch. Using Skeletons in a Java-Based Grid System. In *Euro-Par*, number 2790 in LNCS, 2003.
[3] D. Anderson and *al*. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11), 2002.
[4] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID*, 2004.
[5] N. Andrade, L. Costa, G. Germoglio, and W. Cirne. Peer-to-peer grid computing with the ourgrid community. In *23rd Brazilian Symposium on Computer Networks*, May 2005.
[6] L. Baduel and *al. Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid (chapter 9). Springer, 2006.
[7] F. Baude, D. Caromel, C. Delbé, and L. Henrio. An hybrid message logging-cic protocol for constrained checkpointability. In *Europar 2005*. Springer-Verlag, 2005.
[8] H. Bouziane, C. Pérez, and T. Priol. Modeling and executing master-worker applications in component models. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, april 2006.
[9] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9), 1993.
[10] A. Defrance and *al*. A generic distributed architecture for business computations. application to financial risk analysis. In *2nd International Conference on Distributed Framework for Multimedia Applications*, Penang, Malaysia, May 2006.
[11] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practive*. Pearson Education, 1999.
[12] G.Bronevetsky, D.Marques, K.Pingali, and P.Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, 2003.
[13] J. Gentle. *Statistics and Computing*, chapter Random number generation and Monte Carlo methods. Springer-Verlag, 1998.
[14] P. Glasserman. *Monte Carlo methods in financial engineering*, volume 53 of *Applications of Mathematics*. Springer-Verlag, 2004.
[15] J. Hull. *Options, Futures and Other Derviatives*. Prentice Hall, 2005.
[16] J. Hull. *Risk Management and Financial Institutions*. Prentice Hall, 2006.
[17] D. Lamberton and B. Lapeyre. *Introduction to Stochastic Calculus Applied to Finance*. Chapman and Hall, 1996.