Grid computing for Monte Carlo based intensive calculations in financial derivative pricing applications

Viet Dung Doan

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Nice

2010

Program Authorized to Offer Degree: University of Nice

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Time is gold. This is particularly true in financial markets because every second is passing you can't go back to change previous actions. Any delays are generally unacceptable as the financial market is continually changing in a liquid market (e.g. Chicago Board Options Exchange (CBOE) [1] or Eurex and International Securities Exchange (ISE)[2] the two derivative markets which are the most well-known and actively traded market nowadays). Any decision generally needs to be made in seconds or at most minutes to be useful to traders thus making time constraints is a hot topic for financial institutions. Everyday more complex financial contracts are created in order to enhance the market liquidity. Having more sophisticated products with more complex mathematical models leads the decision making processes to become more computationally intensive. Meanwhile, recent advances in computer technology have led to a proliferation of powerful networked computers that form large distributed systems (e.g. grid computing) to solve advanced computation problems including computational intensive ones. Obviously, financial institutions are the first major leaders that profit from applying high performance computing (HPC) for their needs. Therefore, during the last decade, both academy and industrial works have made every effort to solve such interesting combination of HPC and computational finance.

Particularly, we focus on the derivative market and are interested in the evaluation of high dimensional option contracts, one of the main contract types in the derivative market. As mentioned above, there are more and more sophisticated option contracts with a very large volume of underlying assets that have significant computational requirements. However for high dimensional problems, analytical methods which are very efficient, are not always applicable due to the "curse of dimensionality". The term "curse of dimensionality" due to Bellman (1961) [17] means the exponential growth in volume when adding more extra dimensions in the problem space. For example, the evolution of the option value can be modelled using a partial differential equation (PDE) (e.g. the BlackScholes partial differential equations (PDE) [19]) and a common used numerical method to obtain such option price is the finite difference method which can solve the PDE using a discretized form with tree approaches, see Schwartz (1977) [104]. However because the method presents the evolution of option price using a lattice tree in time and underlying asset price, hence it is

[1]www.cboe.com/

[2]http://www.eurexchange.com

limited whenever increasing the number of underlying assets (increasing the dimension of the problem space). That is why in such high dimensional cases, numerical methods such as Monte Carlo (MC) based methods can be used, see [52]. Another main advantage of the MC method is that its nature is well suited to parallel computing, see [43]. However, to date, although parallel computing for finance is a very active area of research, there has been limited public discussion on the parallelization of computational algorithms, and even less on the reliance on a grid computing environment. The reasons for this are firstly due to the trade secrets that financial institutions have invested in developing their own mathematical models and computing environments to give them advantages over their competitors and secondly because of the difficulties in parallelizazing the quite complex computational algorithms and employing them in a large-size computing environment safely and smoothly.

Research objectives

The research work presented within this thesis had a objective to study the use of HPC for Monte Carlo based methods for computational demanding financial pricing applications. As a consequence, our objectives can be summerized as follows: The first one is a grid programming framework for computational finance which especially targets derivative pricing applications. The second one is the design of implementation of a parallel approach for the Classification Monte Carlo (CMC) algorithm proposed by Picazo (2002) [94] for high dimensional American option pricing.

For the first objective, we aim to provide a framework which includes fault tolerance, load balancing, dynamic task distribution and deployment mechanism for a heterogeneous grid computing environment. This framework should support both either common users who just want a tool to run their applications faster on a grid and the financial software developers who want a programming framework to implement their distributed algorithms and applications. Hence it should abstract both common users and developers from the underlying computing resource infrastructure. For the second objective, we aim to find a parallel algorithm for high dimensional American option pricing that can be scalable in a grid computing environment.

Contributions

In face of those objectives, this thesis presents the following contributions:

- A grid-enabled programming framework for financial applications especially for Monte Carlo based ones (e.g. option pricing and hedging using Monte Carlo methods), named PicsouGrid: PicsouGrid is able to provide fault tolerance, load balancing, dynamic task distribution and deployment on a heterogeneous environment (e.g. a grid environment). It is also designed to support both large Bag-of-Tasks (BoT) problems and parallel algorithms requiring communications.

- A parallel approach for Monte Carlo based algorithm for high dimensional American option pricing: We particularly investigated the Classification–Monte Carlo algorithm for pricing high–dimensional American options. The CMC Algorithm estimates the characterization of the boundary and it can be used for pricing different option types such as maximum, minimum or geometric average basket options using a generic classification configuration. Our parallel approach was scaled up to 64 processors in a grid environment. The experiments were realized with the real world basket option size with a large number of underlying assets (e.g. 40 assets in the CAC40 index[3]). We also analyzed the performance and the accuracy of the CMC algorithm with several classification algorithms such as AdaBoost proposed by Freund and Schapire (1996) [45], Gradient Boost proposed by Friedman (2001) [47] and Support Vector Machines from Cristianini and Shawe-Taylor (2000) [39]. In term of numerical results, using PicsouGrid, we were able to evaluate our option pricing with a large high number of opportunity dates (e.g. up to 100s opportunity dates) in order to obtain the American option prices with high accuracy. Beside, we provided a full explanation and some detailed numerical experiments of the dimension reduction technique which is very useful in validating the estimated option prices that we calculated using CMC algorithm.

- A financial benchmark suite for performance evaluation and analysis of grid middleware: The benchmark suite includes 1000 testcases where each one consists of one high dimensional European option pricing and hedging problem. Such benchmark suite is conceptually simple and easy to understand for both the grid computing and financial communities. For the grid computing community, it is readily redistributable and "generic" to evaluate some specific middleware performance. In the financial context, the benchmark suite includes both algorithms implementation and "reference" results to validate the computed results using provided algorithms in the benchmark suite. The benchmark suite was successfully used in the 2008 SuperQuant Monte Carlo challenge - the Fifth Grid Plugtest and Contest. Within the context of this challenge, we developed the ProActive Monte Carlo API (MC API), a simple programming framework in a distributed environment based upon the ProActive grid

[3]www.euronext.com/

middleware and programming library, that was proved useful for the Grid Plugtest and Contest participants.

Organization of this thesis

This thesis consists of three major contributions, introduced in the previous section. There are five chapters and the content of each chapter is briefly described below:

- Chapter 1 *Context*: In this chapter, firstly in term of computational finance, we will present the mathematical background of the high dimensional option pricing and hedging problems. We also discuss the difficulties appearing when performing high dimensional computations and the adaptable computational methods for such computations such as Monte Carlo based methods. Secondly, we give an overview of grid computing and discuss why Monte Carlo based methods are suited for parallel computing in particularly for grid computing. Finally, we will provide an example of pricing and hedging the high dimensional European options using Monte Carlo methods. Such example is well suited to a Master/Worker parallel computing approach in the context of grid environment.

- Chapter 2 *PicsouGrid: A Grid Framework for Financial Applications*: This chapter will introduce motivations behind PicsouGrid then, resulting architecture, programming supports, proof of concept implementation. The chapter then continues with some experiment of results of PicsouGrid in a grid environment. Using an as simplified as possible option pricing problem, we analyze the performance of PicsouGrid on it, in term of speedup, load balancing and fault tolerance.

- Chapter 3 *Parallel High Dimensional American Option Pricing*: Within this chapter, we will discuss about some American option pricing using Monte Carlo based methods and associated parallel approaches. We particularly focus on the Classification-Monte Carlo (CMC) algorithm proposed by Picazo (2002) [94] for pricing high-dimensional American options. A parallel approach for the CMC algorithm will be provided. Such approach is then successfully evaluated in a computational grid environment using PicsouGrid. We also analyze the performance and the accuracy of the CMC algorithm with several classification algorithms such as AdaBoost, Gradient Boost and Support Vector Machines in order to figure out its performance-accuracy tradeoffs.

- Chapter 4 *Financial Benchmark Suite for Grid Computing* : We will detail the motivations of the SuperQuant Financial Benchmark Suite which we desgined for performance evaluation and analysis of grid middleware in the financial engineering context. We will provide details about how to setuo the values for specific benchmarks, then how to make effectively use it (through a somple programming framework, named ProActive Monte Carlo API) and obtain associated scores to mesure the middleware performance.

- Chapter 5 *Conclusions and Perspectives*: As each of the above chapters already concludes and highlights some specific remaining work if any, this chapter gathers our general conclusions and then discuss about some future perspectives.

Chapter 1
CONTEXT

1.1 Computational Finance

Computational finance (often also known as financial engineering) is an inter-disciplinary field that employs the disciplines of scientific computing, mathematical finance and high performance computing to make financial decisions, as well as facilitating the risk management of those decisions. Utilizing various methods, practitioners of computational finance aim to precisely explore the potential risks that certain financial contracts create. Areas where computational finance techniques are widely employed include investments in stocks and bonds, futures trading, and hedging on stock market activity. Among such areas, financial derivative market plays an important role and occupies a large portion of transaction flow in the global market. For example, according to the Eurex and International Securities Exchange (ISE)[1] annual report, the average daily trading volume of such derivatives markets approximates 10.5 million contracts. Totally, the Eurex and ISE closed out 2009 with a turnover of more than 2.65 billion contracts, compared with 3.17 billion in the record year 2008 due to financial crisis. Similarly, the Chicago Board Options Exchange [2] reported the average daily trading volume of 4.7 million contracts and reached totally 1.193 billion contract in 2009. A derivative is a financial contract whose value depends on (or derives from) the value of other variables (called as the underlying assets). Rather than trade or exchange the underlying asset itself, derivative traders enter into an agreement to exchange cash over time based on the underlying asset. Very often, the underlying assets are the prices of stock. A simple example is a future contract: an agreement to exchange the underlying asset at a future date at a fixed price.

Derivatives are usually broadly categorized by: the relationship between the underlying and the derivative (e.g. forward, option, swap), the type of underlying (e.g. equity derivatives, foreign exchange derivatives, credit derivatives), the market in which they trade (e.g. CBOE, Eurex) or OTC (Over The Counter market)[3]). Contract price must be determined before the trade, with a price that must satisfy both buyer and seller. As we can see, due

[1] http://www.eurexchange.com

[2] www.cboe.com/

[3] trading directly between two parties

to the very large daily trading volume mentioned above determining the fair value for all such derivative contracts with time constraint is really a challenge. We will discuss about this problem later in this chapter.

Determining the fair value for a derivative contract is called: the *pricing* problem. Meanwhile within a derivative contract, there always exists some particular risks that the contract might face up in the future, due to any change of market parameters/conditions. Therefore, in order to handle and reduce such risks in the contract, *hedging* activity is a must. A perfect hedge is one that completely eliminates the market risk. In practice, the hedging impacts will influent the final decision to determine the contract price.

Of course, mathematical models are required for both the pricing and hedging problems. In the particular context of option pricing and hedging, the Cox, Ross and Rubinstein binomial model [37] and the Black-Scholes model [19, 86] are the simplest pricing models for such problem. In this thesis, we consider only the Monte Carlo based methods for multi-dimensional option pricing and hedging which allows us to deal with a wide variety of stochastic model for the underlying asset. A simple but useful model, the multi-dimensional Black-Scholes model therefore has been addressed in this context. Further details of such model will be described in the following section.

1.1.1 Option Pricing and Hedging

An option, one of the main contract used for market risk management, is a contract between a buyer and a seller that gives the buyer the right but not the obligation to purchase (call option) or to sell (put option) a specified amount of a particular underlying asset at a future day at an agreed price. Call and put options both have an exercise price (or strike) K, which is the trading price of the underlying asset fixed by the contract. The option vanishes after a maturity date T. Option contracts are characterized by their time exercise rule and also their payoff type.

We denote $\Psi(S_T, T)$ a general payoff function at time T. As an example, consider an option contract derived from a price S of a stock with maturity date T. A call option has a payoff $\Psi(S_T, T) = \max(S_T - K, 0)$. The zero part corresponds to the scenario where the option holder does not exercise the right to buy. In contrast, a put option with the same K has payoff $\Psi(S_T, T) = \max(K - S_T, 0)$.

Based on the option exercise rule, we can separate options into 3 major categories as follows: An European option can be exercised only at the maturity date T. An American option can be exercised at any time $t \leq T$. A discretized American option is a Bermudan option which can be exercised at a finite set of date $t_m \leq T, m = 1, \ldots, N$.

On the other hand, based on payoff type, we have:

- Vanilla option : a standard call or put option with $\Psi(S_T, T) = \max(S_T - K, 0)$ and $\max(K - S_T, 0)$ respectively.

- Barrier option : an option that is either activated or canceled based on the underlying asset price hitting a certain barrier B. For example, a barrier put option has $\Psi(S_T, T) = \max(K - S_T, 0)\, \mathbb{1}_{\left(\min\limits_{0 \le t \le T}(S_t) \le B\right)}$ will be activated once the asset price hits the barrier B.

- Basket option : an option that bases on a basket of underlying assets. For example, a basket put option has $\Psi(S_T, T) = \max(K - S_T, 0)$ where $S = \{S^1, \dots, S^d\}$ is a basket of d assets price.

- Asian option : an option whose payoff is based on the average of the underlying over some period of time $\{t_m, m = 1, \dots, N\}$. For example, a Asian put option has $\Psi(S_T, T) = \max(K - A_T, 0)$ where $A_T = \dfrac{\sum_{m=1}^{N_T} S_{t_m}}{N}$.

Options whose payoff is calculated differently than vanilla are called "exotic" option. Exotic options can pose challenging problems in pricing and hedging problems because of their complex structures, see Hull (2008) [61].

In the early 1970s, Fischer Black, Myron Scholes and Robert Merton presented their breakthrough model for stock option pricing and it is commonly known as the Black-Scholes (BS) model [19, 86]. The model assumes that the price of an asset follows a geometric Brownian Motion (1.1), Samuelson (1965) [101], with constant drift and volatility.

$$S_t = S_0 + \int_0^t \mu S_s d_s + \int_0^t \sigma S_s dW_s \tag{1.1}$$

here t is the time of observation, μ is the mean rate of return (in percentage) of asset, σ is the volatility rate (in percentage) of the asset and W_t is a one-dimensional Brownian motion. The BS model also makes the following explicit assumptions:

- There is no arbitrage opportunity. For example in a stock market, an arbitrage opportunity can be simply seen as an opportunity to buy a stock share at a low price then immediately selling it for a higher price on a different market, thus making a risk-free profit.

- S_t is a continuous process in time t.

- The interest rate is constant at any time.

- There are no transaction costs.
- In case of stock, it does not pay a dividend.
- There are no restrictions on *short selling*, where *short selling* involves selling a stock that is not owned.

The BS model has had a huge influence in financial markets in general and particularly the derivative market. The multi-dimensional Black-Scholes model is an extension of the BS model for multi-dimensional option pricing problems. For example, consider a basket of d underlying asset that is described using the multi-dimensional Black-Scholes model in (1.2). Such a multi-dimensional model is a simple model to describe the evolution of a basket of d underlying assets price through a system of stochastic differential equations (SDEs) [71],

$$dS_t^i = S_t^i(r - \delta_i)dt + S_t^i \sigma_i dB_t^i, \; i = 1, \ldots, d, \text{ where} \tag{1.2}$$

- $S = \{S^1, \ldots, S^d\}$ is a basket of d assets price.
- r is the constant interest rate for every maturity date and at any time.
- $\delta = \{\delta_1, \ldots, \delta_d\}$ is a constant dividend vector.
- $B = \{B^1, \ldots, B^d\}$ is a correlated d-dimensional Brownian Motion (BM).
- $\sigma = \{\sigma_1, \ldots, \sigma_d\}$ is a constant volatility vector.

Fundamentally, the option pricing relies on the arbitrage pricing under risk-neutrality. Within a market without arbitrage opportunity, if two contracts have the same payoffs, they must have the same market price: the arbitrage price. Moreover, 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. We denote B_t, the bond value, continuously compounded then $B_t^* = e^{rt}$ in the BS model for instance, where r is the constant 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 [71, 61] states that the fair price at time zero of an option is the expected value, under $\mathbb{P}^*$, of its discounted payoff. Hence, the fair price V for an European option contract is given by the following expression

$$V(S_0, 0) = \mathbb{E}\left[e^{-rT}\Psi(S_t, t \in [0, T])\right]. \tag{1.3}$$

In case of American option pricing, an American option price must at least equal its payoff value otherwise there would be an arbitrage opportunity. For example, assume an American option on an underlying asset S has a price less than its payoff value, $V(S_t, t) < (K - S_t)$. One can buy such put contract for V and also buy the underlying asset for S, then immediately exercise the put contract for K, thus made a risk-free profit $K - S - V > 0$. In order to avoid such arbitrage opportunity, we have:

$$V(S_t, t) \geq \Psi(S_t, t) \tag{1.4}$$

At any time, the underlying asset price where we should exercise the American option is called the optimal exercise point. During the option life T, these optimal points will form a continuous boundary which depends on time, called the optimal exercise boundary $b^*(t) = \{\mathrm{x}; V(\mathrm{x}, t) = \Psi(\mathrm{x}, t)\}$. Such boundary divides the space of an American option into two regions:

- the continuation region, $\mathcal{C}(t) = \{\mathbf{x}; V(\mathbf{x}t) > \Psi(\mathbf{x}t)\}$, where one should hold the option contract rather than exercise it early.

- the stopping region, $\mathcal{S}(t) = \{\mathbf{x}; V(\mathbf{x}, t) \leq \Psi(\mathbf{x}, t)\}$, where one should exercise the option immediately.

There are several ways to formulate the American option pricing problem [126]. A common way is the optimal stopping formulation where the fair price at time zero of an American option can be written as,

$$V(S_0, 0) = \sup_{\tau} \mathbb{E}\big[e^{-r\tau}\Psi(S_\tau, \tau \in [0, T])\big]. \tag{1.5}$$

where the maximum is taken over all stopping time $\tau \in [0, T]$. This supremum is achieved by an optimal stopping time τ^* under the form

$$\tau^* = \inf\{t \geq 0 : S_t \in \mathbb{S}\} \tag{1.6}$$

Though we address the simulation methods for American option pricing, we restrict ourselves to American options that only can be exercised at a finite set of opportunity dates $\Theta = \{t_m, m = 1, \ldots, N\}$, then in this case such options become Bermudan options. Of course, one would like to approximate the American price by letting m increase to infinity. Future references to American options imply both American and Bermudan options. Then the American option price at any time t_m can be approximated on a discrete set of opportunity date as follow,

$$V(S_{t_m}, t_m) = \sup_{\tau \in \Theta} \mathbb{E}\left[e^{-r\tau}\Psi(S_\tau, \tau)|S_{t_m}\right]. \tag{1.7}$$

The notation $\mathbb{E}(Y|X)$ means the expectation of Y knowing X. To get the option price at time zero $V(S_0, 0)$, Equation (1.7) can be rewritten under the dynamic programming representation as follows:

$$\begin{aligned} V(S_{t_N}, t_N) &= \Psi(S_{t_N}, t_N) \\ V(S_{t_m}, t_m) &= \max\Big(\Psi(S_{t_m}, t_m), \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big] \end{aligned} \tag{1.8}$$

Equation (1.8) shows that at time t_m the option holder should exercise the option whenever $\Psi(S_{t_m}, t_m) > \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]$ and to hold it otherwise. We define the term $\Psi(S_{t_m}, t_m)$ as the exercise value (also called intrinsic value) of the option and the other term $\mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]$ as the continuation value.

Many methods for American option pricing rely on the dynamic programming representation in Equation (1.8). The main difficulty is the estimation of the conditional expectation in Equation (1.8). Such methods will be introduced later in this section.

As mentioned earlier, along with option pricing, option hedging (also called Greek hedging) is a very important activity. The Greeks represent sensitivities of option price with respect to the change of market parameters like asset price, time remaining to maturity, volatility, or interest rate. As an example, the Greek delta (Δ_t) is the first derivative of the option price V with respect to the change of asset price S, which is defined as:

$$\Delta_t = \frac{\partial V(S_t, t)}{\partial S_t} \tag{1.9}$$

Normally, Greeks are not observed in the real time market but, provides information that needs to be computed with accuracy for determining the hedge of any derivative contract. There are several common used Greeks such as Gamma (Γ), Rho (ρ) and Theta (θ). We refer to [127, 61] for complete explanations of these Greeks.

In practice, usually Greeks are derivatives of first or second order that can be computed using finite difference methods [52]. First and second order derivative are approximated respectively as follows:

$$\begin{aligned} \text{Greek}^{(1)}(x) &\simeq \frac{V(S_0,0)|_{x+\epsilon_x} - V(S_0,0)|_{x-\epsilon_x}}{2x\epsilon_x} \\ \text{Greek}^{(2)}(x) &\simeq \frac{V(S_0,0)|_{x+\epsilon_x} - 2V(S_0,0) + V(S_0,0)|_{x-\epsilon_x}}{(x\epsilon_x)^2} \end{aligned}$$

where x is one parameter among the market parameters mentioned above. The value $V(S_0, 0)|_{x \pm \epsilon_x}$ the option prices with respect to the change of parameter x, with a small ϵ_x value.

In this paragraph, we are going to discuss about the approximation methods for option pricing and hedging. To date, many approaches have been used to address these problems, see [127, 52, 61]

Explicit Formulas: When the model is simple (e.g. one dimension; the volatility rate is constant; stock without paying dividend), we can obtain the explicit formula to calculate the derivative price. For example, in Black and Scholes (1973) [19] and Merton (1973) [86], authors provided a closed form solution for a European option. Rubinstein and Reiner (1991a) [100] defined an analytical model for the value of European-style barrier options on stock. It is possible to value perpetual American put option in closed form, see Merton, (1973) [86] and Wilmott (1998) [127]. The perpetual American option is the simplest but still interesting example of American option because its optimal exercise rule is not obvious and can be determined explicitly, see Shreve (2004) [105]. However, when the model becomes more complex, for example to accommodate multi-dimensional problems, there has been very limited research into analytic solutions for such multi-dimensional option pricing. Then numerical methods for approximation and simulation are required (e.g. Monte Carlo based methods). In fact, for multi-dimensional European option pricing, it is enough to simulate the underlying asset prices at the maturity date S_T to estimate the option price. Meanwhile, for American options or path dependent options such as barrier ones, the common background of pricing methods is to simulate the paths of evolution of the underlying assets $\{S_t, t \in [0, T]\}$ and then use these paths to estimate the option price. Some widely used methods for option pricing and hedging in practice are briefly introduced in the following list:

- **Tree and Lattice Methods** : The binomial model was first proposed by Cox, Ross and Rubinstein (1979) [37]. The idea of the model is to use a "discrete-time framework" to trace the evolution of the underlying asset price (e.g. stock) via a binomial tree, for a given number of time steps between valuation date and option expiration. Once having the entire tree of asset prices at all steps, starting at the final nodes of the tree, the option price can be obtained by a backward computing on the tree. These methods can be used for both European and American option pricing. Notice that in case of American option pricing, before the appearance of the Longstaff and Schwartz (2001) [77] method, the binomial method was the only solution for this class of option pricing. However, for European option pricing, the binomial model is very slow compared with other methods. Moreover, its efficiency with regard to speed is strongly dependant upon the option type. For example, regarding the options with complicated payoff such as barrier, Asian options such methods present a slow and irregular convergence behaviour, see Korn (2001) [67].

There also exists two other alternative approaches for American option pricing based on the trees methods. The first one is the **Stochastic Tree Methods**. Broadie and Glasserman (1997a) [25] suggest using a bushy tree to estimate an American option price. However, the algorithm complexity grows exponentially with the number of exercise opportunities, so such a method is applicable only when the number of exercise opportunities is small (e.g. smaller than 5, see Glasserman (2004) [52]). Recently Tomek (2006) [118] modified this approach to limit the computational burden.

The second one is the **Stochastic Mesh Methods**. The stochastic mesh method proposed by Broadie and Glasserman (2004) [26] for American option pricing can be thought of as a recombining stochastic tree. The idea is to produce a mesh rather than a tree. That keeps the number of nodes at each exercise opportunity fixed, hence avoiding the exponential complexity growth of a tree.

This method is effective for solving high dimensional problem with many exercise opportunity dates. However, the computational complexity is $\mathbb{O}(b^2)$ where b is the number of simulated paths from the seed of the tree.

- **Partial Differential Equation (PDE)** : The method was first applied to option pricing by Schwartz (1977) [104]. Such PDE method obtains the option price by solving the partial differential equation that the option price is a solution using finite difference methods (FD). Mathematically, the American option pricing problem forms a free boundary problem of the Black-Scholes equation, for which the free boundary corresponds to the optimal exercise boundary. Consider the option price $V(S_t, t)$ solves the Black-Scholes equation:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 B}{\partial S^2} - rV = 0 \tag{1.10}$$

with boundary conditions

$$V(\mathbf{x}, T) = \Psi(\mathbf{x}, T) \tag{1.11}$$

$$V(\mathbf{x}, t) = \Psi(\mathbf{x}, t) > 0, \ \forall \mathbf{x} \in b^*(t) \tag{1.12}$$

$$\frac{\partial V}{\partial S}(\mathbf{x}, t) = 1, \ \forall \mathbf{x} \in b^*(t) \tag{1.13}$$

Then $b^*(t)$ represents the optimal exercise boundary at time t, called optimal exercise boundary

$$b^*(t) = \{\mathbf{x} : V(\mathbf{x}, t) = \Psi(\mathbf{x}, t)\} \tag{1.14}$$

The FD methods work on a grid whose horizontal axis represents time and vertical axis represent the underlying asset price. The FD methods are very efficient for option pricing with up to 3 underlying assets, they offer fast convergence and easy computation of the sensitivities (Greek hedging). But for multi-dimensional problems, this method is often prohibitively computationally demanding and suffers from the "curse of dimensionality ": standard discretization of the PDE leads to systems that grow exponentially with the dimension of the problem.

Recently some techniques for FD methods have been proposed to handle such multi-dimensional problems. For example, the sparse grid method is proposed by Reisinger (2004) [97] for both European and American option pricing. This method divided the main problem on several sparse grids, then solves these sub-problems which have the same spatial dimensionality as the original problem but coarser discretization, and finally properly combines the partial solutions to get the final one. Another advantage of the sparse grids method is that it builds up a hierarchy of grids that can be solved in parallel. However, such method is only useful in with a medium dimension d ($4 \leq d \leq 10$) see Achdou and Pironneau (2005) [2].

- **Quantization Method** : A method proposed by Bally and Pagès (2005) [14] that uses a finite-state dynamic program to approximate the price of an American option. While the dynamic program in random tree based methods is based on randomly sample states, in quantization method, such states are defined in advance based on the partitioning of the state space of the Markov chain. Bally and Pagès (2005) [14] discussed how to select an optimal state space partition and also proposed simulation based procedures for such partitions. However, these simulated partitions above are not well suited to high dimensional state space (e.g. with a large number of underlying assets). This method is insensitive to the number of opportunity dates but it is inapplicable to high dimensional problems.

- **Monte Carlo Based Methods (MC)** : While almost all numerical methods for multi-dimensional pricing problems are limited in terms of the number of dimensions, MC methods for option pricing are usually preferred because it can naturally handle such problems. Another advantages of MC based methods are that they easily accommodate options with complex payoff functions (e.g. Asian options and lookback options are two typical examples of path dependent options) and are well suited with parallel computing. Most of common MC based methods for American option pricing are detailed in Glasserman (2004) [52] and Tavella (2002) [113]. A comparison of some widely used MC based methods for American option pricing was made in Fu and al. (2001) [49]. In term of Greeks hedging, Monter and Kohatsu-Higa (2003) [89] proposed the use of Malliavin calculus to estimate the Greeks using Monte Carlo method. We can also use the FD methods with the MC simulations for Greeks hedging. The main limitation of MC methods is the slow rate of convergence which is $\mathbb{O}(1/\sqrt{n})$, where n is the number of MC simulations. However, this limitation can be overcome by applying the parallel computing techniques. This is one of the main research issues in this thesis.

- **Quasi Monte Carlo Methods (QMC)** : QMC methods have been shown to be very useful in finance. Chaudhary (2005) [32] and Lemieux (2004) [76] have applied sequences of quasi-random numbers to the valuation of American options. However, the effectiveness of QMC methods in high dimensional pricing problems is not steady. In fact, QMC methods have the convergence rate ranging from the $\mathcal{O}((\log n)^d/n)$ (where n is the number of quasi-random points and d is the number of dimensions) to nearly the $\mathcal{O}(1/n)$ rate. With the $\mathcal{O}((\log n)^d/n)$ rate, for d small the rate of convergence of QMC is faster than MC $\mathcal{O}(1/\sqrt{n})$ but not for a large d. Bratley and al. (1992) [23] performed some tests on certain mathematical problems and concluded that in high dimensional problem, say $d > 12$, QMC seems to offer no practical advantages over Monte Carlo methods. Thus, in finance, it was known that QMC should not be used for high dimensional problems, see discussions in Glasserman (2004) [52], Jackel (2002) [64] and Tavella and Randall (2000) [114] for more details.

Based on this brief overview of numerical methods for option pricing and hedging above, it is clear that when the number of dimensions in the problem is large, most of numerical methods become intractable, and in these cases Monte Carlo methods often give better results. In the next paragraphs, we are going to introduce the use of Monte Carlo methods for high dimensional pricing and hedging problems. However, within the context of this chapter, we only illustrate such problem though an example of basket European option pricing and hedging. Not like an European option, an American option on the other hand may be exercised at any time before the expiry date that make pricing and hedging an American option more complicated and difficult. Therefore, further discussions about the problem formulation and pricing algorithms (e.g. Monte Carlo based algorithms) for high dimensional American will be discussed carefully in Chapter 3.

1.1.2 European Option Pricing using Monte Carlo Methods

In order to approximate the expectation in (1.3), the Monte Carlo method consists in computing the arithmetic mean of $nbMC$ independent pseudo random simulations of the option payoff $\Psi\big(f(S_t,t\big)$, according to the stochastic model (1.2). We have such that

$$V(S_0,0) \simeq \frac{1}{nbMC}\sum_{s=1}^{nbMC} e^{-rT}\Psi\big(f(S_t^{(s)},t\in[0,T])\big) \tag{1.15}$$

where $S_t^{(s)}$ are independent trajectories of the solution of (1.2) and $nbMC$ is the number of Monte Carlo simulations. Regarding the BS model described in (1.2), in order to simulate $S_t^{(s)}$, we complete the model description with the correlation matrix $\big(\rho_{ij},\ i,j=1,\dots,d\big)$ of the Brownian Motion B, such that $\rho_{ij}=\frac{\mathbb{E}(B_t^i,B_t^j)}{t}$ so $Covariance(B_t^i,B_t^j)=\rho_{ij}t$. The calibration of ρ_{ij} will be discussed later in Chapter 4. We define the $d\times d$ covariance matrix Cov by,

$$Cov_{ij}=\sigma_i\sigma_j\rho_{ij}. \tag{1.16}$$

We aim to rewrite equation (1.2) by the following equation (1.17),

$$dS_t^i = S_t^i r dt + S_t^i\sum_{k=1}^{d} a_{ik}dW_t^k \tag{1.17}$$

where $\big(a_{ik},i,k=1,\dots,d\big)=A$, such that $AA^t=Cov$, thus

$$Cov_{ij}=\sum_{k=1}^{d}a_{ik}a_{jk},\ \ i,j=1,\dots,d \tag{1.18}$$

and $W=\big(W^1,\dots,W^d\big)$ is a standard d-dimensional Brownian Motion, where each W^i is independent to each other. Note that A exists, as Cov is always a positive–definite matrix. By applying Itô Lemma [71] for (1.17), we have

$$S_t^i = S_0^i\exp\big((r-\frac{1}{2}\sigma_i^2)t+\sum_{k=1}^{d}a_{ik}dW_t^k\big)$$

A trajectory realization of asset prices at discrete opportunities in Θ set is obtained by setting

$$S_{t_{m+1}}^i = S_{t_m}^i\exp\big((r-\frac{1}{2}\sigma_i^2)(t_{m+1}-t_m)+\sqrt{(t_{m+1}-t_m)}\sum_{k=1}^{d}a_{ik}Z_{t_{m+1}}^k\big) \tag{1.19}$$

where $\big(Z_m = \big(Z_m^1, \dots, Z_m^d\big)\big)$ is a family of independent Gaussian variables of law $\mathcal{N}(0, Id)$.

To illustrate an option pricing application using MC methods, we provide the following pseudo–code for pricing a call Geometric Average (GA) European option of d independent assets in Algorithm 1. The GA option has payoff $\Psi(f(S_t), t)$ with $f(x) = \prod_{i=1}^{d} x_i^{\frac{1}{d}}$. The European type options only consider the payoff at maturity date T, $\Psi(f(S_T), T)$. And since the assets are independent (there are no correlation between assets), we can simplify the trajectory realization of asset prices in Equation (1.20) as follows:

$$S_T^i = S_0^i \exp\big((r - \frac{1}{2}\sigma_i^2)T + \sqrt{T}\sigma_i Z_T^i\big) \tag{1.20}$$

We simulate $Z_T^{(s)} = \{(Z_T^{1,(s)}, \dots, Z_T^{d,(s)}), s = 1, \dots, nbMC\}$, a sequence of independent Gaussian $\mathcal{N}(0, T)$ trials. We can transform the uniform random trials generated by a pseudo-random number generator (PRNG) to normal distributed ones by using the common Box-Muller method (1958) [20].

In fact, the Monte Carlo method relies on the use of a sequence of pseudo-random numbers to produce its simulated results. In fact, a true random number is only generated by hardware random number generators or a physical process like a dice throw for example. However, in practice we use algorithms to generate the numbers that approximates the properties of true random number, called pseudo-random numbers. The sequence of an ideal PRNG should: be uniformly distributed; not be correlated; have long period; be reproducible; be fast; be portable; require limited memory and pass tests for randomness. In fact, a serial PRNG can satisfy most of such qualities. Common used serial PRNGs are linear congruential generators, lagged Fibonacci generators etc, see [66, 73].

The Law of Large Number of Monte Carlo methods implies that

$$\lim_{nbMC\to\infty} \frac{1}{nbMC} \sum_{j=1}^{nbMC} e^{-rT}\Psi\big(f(S_t^{(j)}, t \in [0,T])\big) \equiv V(S_0, 0) \tag{1.21}$$

with the probability 1. The Law of Large Numbers explained the convergence of the MC method. Meanwhile, the Central Limit Theorem gives us the rate of convergence that we shortly describe here as:

$$\lim_{nbMC\to\infty} \frac{\sqrt{nbMC}}{\widehat{s}_C}\big(\mathbb{E}[C_j] - V(S_0, 0)\big) = 0, \text{ with } j = 1, \dots, nbMC. \tag{1.22}$$

where the estimator $\widehat{s}_C$ for the standard deviation of '$V(S_0, 0)$ is computed as

$$\widehat{s}_C = \sqrt{\Big(\frac{1}{nbMC}\sum_{j=1}^{nbMC} C_j^2\Big) - \Big(\frac{1}{nbMC}\sum_{j=1}^{nbMC} C_j\Big)^2}.$$

Thus, the option price $V(S_0, 0)$ is obtained with a 95% confidence within the following interval $\big[\pm 1.96\frac{\widehat{s}_C}{\sqrt{nbMC}}\big]$. The rate of convergence is $\frac{\widehat{s}_C}{\sqrt{nbMC}}$. Hence, for example to get a rate of order 10^3, we need to simulate at least $nbMC = \frac{10^6}{\widehat{s}_C^2}$ simulations.

Considering the Greeks hedging for the GA European option pricing in Algorithm 1 above, let us denote $V(S_0, 0)|_{S_0^i \pm \epsilon_S}$ the option prices with respect to the change of the asset price S^i, $V(S_0, 0)|_{r \pm \epsilon_r}$ and $V(S_0, 0)|_{\tau \pm \epsilon_\tau}$ the option prices with respect to the change of the interest rate r and of the time remained to maturity τ. Algorithm 2 below presents the pseudo code for the Greeks hedging by using finite difference methods. The vector of Δ and the matrix of Γ respectively are the first and second order derivatives of $V(S_0, 0)$ with respect to the change of each asset price among the basket. To simplify the computation, we only compute the diagonal of the matrix Γ. The two last Greeks ρ and θ are the first order derivatives of $V(S_0, 0)$ with respect to the change of r and τ. As observed in Algorithm 1, the simulations for Greeks evaluation can use the same random numbers than the ones for option price evaluation.

Based on the computation of Greeks hedging presented in Algorithm 2, we can observe that the Greeks values depend on the option prices with or without respect to the change of any parameter. Obviously, once got these accurate enough option prices, we can achieve good Greek values respectively. For example, let assume that the ϵ_S is in order of 10^{-2}. Hence in order to obtain the Δ^i with the precision 10^{-2}, the option price $V(S_0, 0)|_{S_0^i \pm \epsilon_S}$ must be obtained with the precision 10^{-4} which requires 10^8 number of MC simulations.

Interestingly, the number of simulations grows with the statistical precision seeked for option pricing and hedging, proportionally to $nbMC$. 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. Moreover, once considering Greek hedging for options, the entire Monte Carlo simulations must be performed for each small change in input parameters. That increases the computational time (especially in case of more complex problems such as high dimensional American options pricing and hedging). Evidently, this observation of computational time is discussing in the context of pricing an individual option pricing. As we mentioned earlier in the beginning of this chapter, the daily trading volume of option contracts of the two world largest financial derivative markets CBOE and Eurex reached to millions. Therefore, getting both

Algorithm 1 Pricing and hedging a call Geometric Avarage European option of d assets

```
Require: $S_0^i$, $d$, $r$, $\delta_i$, $\sigma_i$, $T$, $N_T$ and number of simulations $nbMC$
 1: for $s = 1$ to $nbMC$ do
 2:     $f(S_T) = 1$
 3:     $f(S_T^\Delta) = 1$
 4:     for $i = 1$ to $d$ do
 5:         Simulate $Z_T^{i,(s)}$
 6:         // for option pricing
 7:         $S_T^i = S_0^i \exp\Big(\big((r-\delta_i)-\sigma_i^2/2\big)T + \sqrt{T}\sigma_i Z_T^{i,(s)}\Big)$
 8:         $f(S_T) = f(S_T) \times S_T^i$
 9:         // prepare for delta hedging
10:         $S_T^{i,\Delta} = (S_0^i \pm \epsilon_S)\exp\Big(\big((r-\delta_i)-\sigma_i^2/2\big)T + \sqrt{T}\sigma_i Z_T^{i,(s)}\Big)$
11:         $f(S_T^\Delta) = f(S_T^\Delta) \times S_T^{i,\Delta}$
12:         Similary compute $S_T^{i,\Gamma}, S_T^{i,\rho}, S_T^{i,\theta}$
13:     end for
14:     // for option pricing
15:     $f(S_T) = \sqrt[d]{f(S_T)}$
16:     $C_j = e^{-rT}\big(f(S_T) - K\big)^+$
17:     // In order to calculate the variance, we also compute ${C_j}^2$
18:     ${C_j}^2 = \Big(e^{-rT}\big(f(S_T) - K\big)^+\Big)^2$
19:     // prepare for delta hedging
20:     $f(S_T^\Delta) = \sqrt[d]{f(S_T^\Delta)}$
21:     $C_j^\Delta = e^{-rT}\big(f(S_T^\Delta) - K\big)^+$
22:     Similary compute $C_j^\Gamma, C_j^\rho, C_j^\theta$
23: end for
24: // for option pricing
25: return $\widehat{C} = \dfrac{C_1 + \cdots + C_{nbMC}}{nbMC} \equiv V(S_0, 0)$
26: return $\widehat{C}^2 = \dfrac{C_1^2 + \cdots + C_{nbMC}^2}{nbMC}$
27: // prepare for delta hedging
28: return $\widehat{C}^\Delta = \dfrac{C_1^\Delta + \cdots + C_{nbMC}^\Delta}{nbMC} \equiv V(S_0, 0))|_{S_0^i \pm \epsilon_S}$
29: return $\widehat{C}^\Gamma, \widehat{C}^\rho, \widehat{C}^\theta$
```

Algorithm 2 Delta, Gamma, Rho and Theta hedging for a call GA option of d assets

```
Require: $V(S_0,0)$, $V(S_0,0)|_{S_0^i \pm \epsilon_S}$, $V(S_0,0)|_{r \pm \epsilon_r}$ and $V(S_0,0)|_{\tau \pm \epsilon_\tau}$
Ensure: $\Delta$, $\Gamma$, $\rho$, $\theta$
1: for $i = 1$ to $d$ do
2:     $\Delta^i = \dfrac{V(S_0,0)|_{S_0^i+\epsilon_S} - V(S_0,0)|_{S_0^i-\epsilon_S}}{2S_0^i\epsilon_S}$
3:     $\Gamma^{ii} = \dfrac{V(S_0,0)|_{S_0^i+\epsilon_S} - 2V(S_0,0) + V(S_0,0)|_{S_0^i-\epsilon_S}}{(S_0^i\epsilon_S)^2}$
4: end for
5: $\rho = \dfrac{V(S_0,0)|_{r+\epsilon_r} - V(S_0,0)|_{r-\epsilon_r}}{2r\epsilon_r}$
6: $\theta = \dfrac{V(S_0,0)|_{\tau+\epsilon_\tau} - V(S_0,0)|_{\tau-\epsilon_\tau}}{2\tau\epsilon_\tau}$
7: return $\Delta$, $\Gamma$, $\rho$, $\theta$
```

the fast pricing time for such a number of contracts and the price precision is really a challenge in term of critical time constraint. However, the nature of independent Monte Carlo simulations provides significant opportunities of parallelism. The number of simulation, for a desired precision, can be divided into many smaller groups of simulations and exercised concurrently on a parallel computing infrastructure. And this is a clear evidence that the grid computing power could serve and bring some significant advantages and capacity of innovation for such pricing and hedging problems such as achieving a required option price precision and within a short delay.

1.2 Grid Computing

Known as father of grid computing, Ian Foster proposed an innovative vision of networked computers in 1998 [44] that: “A computational grid is a hardware and software infrastructure that proves dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities”. Hence we can see that, a computing grid is the combination of computer resources from multiple administrative domains applied to solve a scientific, technical or business problem that requires a great number of computer processing cycles and/or the need to process large amounts of data.

One of the main strategies of a grid is to harness the computing power of networked computing resources relying on software to divide and deploy pieces of a program among several computers, sometimes up to many thousands. The networked computing resources can be highly distributed, constituting of compute clusters, as well as network distributed parallel processing entities (e.g. multi-core CPUs and parallel machines etc.) thus forming a heterogeneous environment. The size of a grid may vary from a small confined network of computer workstations within a corporation, to a large one as the results of public collaboration across many companies and public institutions.

Grid computing has been around for over a decade now and its advantages are numerous. Some advantages are quite obvious:

1. It is economic. No need to buy expensive computing servers dedicated for hosting a single (private) application but relying on sharing instead.

2. Much more efficient use of idle computing resources (e.g. desktops, workstations). Many of these resources sit idle especially during off business hours.

3. grid environments are much more modular and don't have single points of failure. If one of the servers/desktops within the grid fails there are plenty of other resources able to pick the load. Jobs can more or less automatically restart if a failure occurs.

4. This model scales very well as by nature, the grid is built around a decentralized architecture. A new computing resource is easy to plug in by installing a grid client. A resource can be removed just as easily on the fly.

5. Upgrading can be done on the fly without scheduling downtime of the whole infrastructure. Since there are so many resources, some can be taken offline while leaving enough for work to continue.

6. grid environments are extremely well suited to run jobs that can be split into smaller and independent tasks and run concurrently on many nodes. Hence jobs can be executed in parallel reducing overall waiting time for results.

In general, a grid application is often constructed with the aid of general-purpose grid software libraries and middleware. The distributed nature of grid computing should ideally be kept transparent to the user. When a user submits a job it does not have to think about which machine the job is going to get executed on. The grid software libraries and middleware will perform the necessary calculations and decide where to send the job based on policies.

In contrast, grid computing presents some disadvantages such as the need to have fast interconnect between computing resources; software licensing across many clusters; the need of appropriate tools for managing a large heterogeneous environment; security etc.

As an example, the Enabling Grids for E-science (EGEE) is Europe's leading grid computing project. The main goals of the project aim to contribute recent advances in grid technology and provide an available grid infrastructure to scientists across Europe, with continuous availability. Hence it focuses on expanding and optimizing the grid infrastructure of EGEE to support academic and public sector research user communities. Currently, the EGEE project provides a computing support infrastructure for over 10000 researchers world-wide, from all basic sciences (e.g. high energy physics, earth sciences, life sciences.). Additionally, several business applications from medicine [53, 88], geophysics (e.g. Geocluster[4]) have been deployed on EGEE grid.

In finance, applications such as option pricing and hedging (mentioned above), risk analysis and portfolio management all process a very large number of computational tasks to provide results. Grid computing thus can provide the ability to perform thousands of computations or transactions at once, and in doing so can offer a great opportunity to the financial industry to improve their performance and, therefore, profitability. Nowadays, in practice financial institutions are more and more using parallel computing (including grid computing) for the time critical and large-volume computations with their in-house large-size high performance computing system (e.g. up to thousands computation cores).

Exploring the relationship between grid computing and business applications, BeInGrid (Business Experiments in Grid) is another European Union project (starting at June, 2006 - ending December, 2009) whose objective is to understand the requirements for grid use in commercial environments (e.g. software vendors, IT integrators, service providers and end-users) and to enable and validate the adoption of grid technologies by business. The outcomes of the project are numerous success stories from the use of grid solutions by the businesses parties involved in the project. The use of grid computing in financial applications includes success stories such as enhancing system performance, computing large data sets, calculating risks.

Similarly but focusing on solving mathematical finance problems, the ANR French funded project GCPMF "Grille de Calcul Pour les Mathématiques Financières" was setup in order to advance research on the use of parallel computing for financial application involving both technology companies and research institutions. In particular, the project focused on studying the suitability for finance of high performance hardware architecture (e.g. Graphics Processing Units (GPUs), clusters, grid etc), grid middleware (e.g. ProActive [29]), grid-enabled software (e.g parallel NSP/Premia [31], PriceIt[5]) and revised on new parallel algorithms [80, 42] for mathematical finance problems. This research work happened in this context.

[4]http://www.cggveritas.com

[5]http://www.pricingpartners.com/

1.2.1 Master/Worker Pattern

In grid computing, the Master/Worker pattern is also one of the core patterns for parallelizing work in grid computing. It is well suited with grid technologies and has been widely used in grid-enabled applications. In this section, we will explain the characteristics of such pattern and also discuss about some possible implementation strategies.

As mentioned above, the Master/Worker pattern is a good approach for resolving many parallel problems. This pattern is summarized in Figure 1.1. The solution relies on two or more logical elements: a master and one or more instances of a worker. The master initiates the computation and sets up the problem. It then creates tasks. A task can be distributed to workers based on a static uniform distribution (each worker only receives one task) in the case of a homogeneous environment. In the case of a heterogeneous environment, the master should consider a *dynamic* distribution mechanism where each worker grabs a given number of tasks, carries out the work, and then goes back to grab more tasks. Such *dynamic* distribution has an advantage that the fastest processors can serve the most, but requiring a worker performance analysis mechanism which is very difficult and complicated in grid environment to accurately estimate task size as well as task numbers. In term of task collection, the master then waits until all tasks are done, merges the results, and then shuts down the computation.

In some particular cases, the master needs to run *at least* a fixed number of simulations (e.g. Monte Carlo simulations) to achieve the required results. Instead of distributing such fixed number of simulations uniformly, we can create a large number of tasks (each includes a very few number of simulations). Then a solution, we call *aggressive* distribution, consists in sending such tasks to workers until the master has collected enough simulation results (not just until all tasks have been sent to the workers). Then, useless tasks which are still running on workers can be cancelled. Such a distribution has an advantage that the fastest processors can serve the most, without requiring performance analysis like the *dynamic* distribution.

In the next section, we are going to present a simple may be naive example to illustrate that Monte Carlo principles are well adapted to a parallel environment relying on a Master/Worker pattern.

1.2.2 Monte Carlo Based Methods for Parallel Computing

We consider the estimation of Π using Monte Carlo method, a trivial example just to illustrate that Monte Carlo principles are well adapted to a parallel environment relying on a Master/Worker pattern. Here are some useful facts:

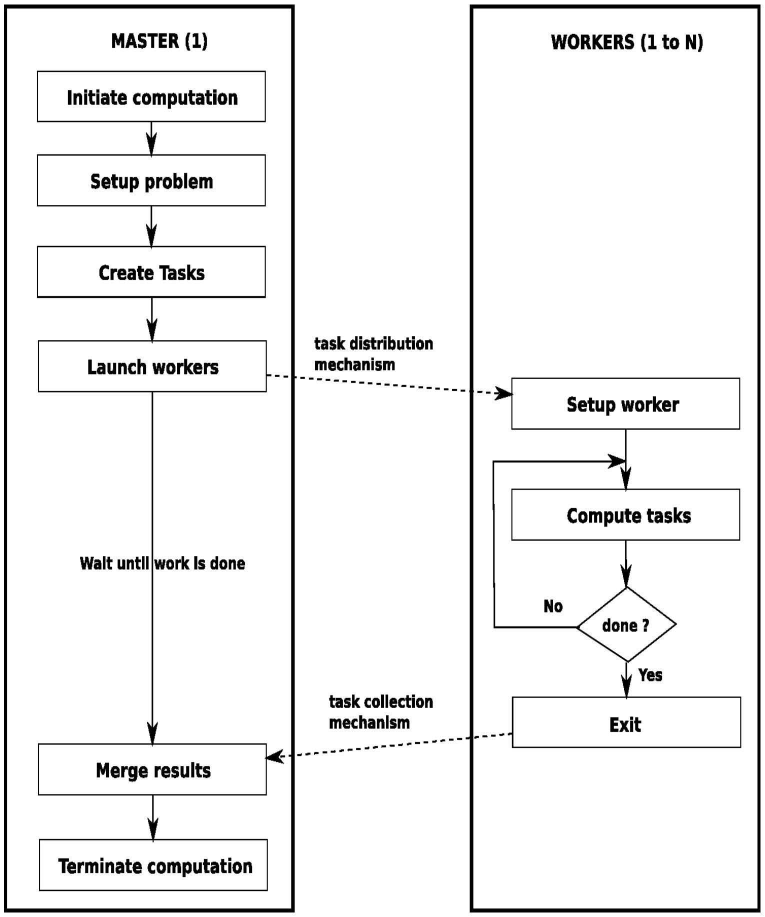

Figure 1.1: Master/Worker Pattern

- The area of a circle is equal to Π times the square of the radius.
- The area of a square is equal to the square of the length of the side.

Imagine a quadrant of circle inscribed inside a unit square. Thus the radius of the circle is $R = 1$ and the quadrant of circle area is $\frac{1}{4}\Pi R^2 = \frac{\Pi}{4}$. The square's area is $R^2 = 1$. Therefore the ratio of the area of the circle to the area of the square is $\frac{\Pi}{4}$. Using a random number generator, we can fulfil standard uniform random numbers U(0,1) at the square. The ratio between the number of points that fall inside the circle and the total number of points that fall inside the square is an approximation to the value of $\frac{\Pi}{4}$ as follows

$$\frac{\Pi}{4} = \int_0^1 \int_0^1 g(x,y) f(x,y) dx dy \tag{1.23}$$

where $\{x, y\}$ are random points coordinates in the range $[0, 1]$ and

$f(x, y) = 1$ because the area of the unit square $= 1$ and

$$g(x,y) = \begin{cases} 1, & \text{when } x^2 + y^2 \leq 1, \text{ considering a point given by } \{x, y\} \text{ is inside the quadrant} \\ 0, & \text{when } x^2 + y^2 > 1, \text{ otherwise.} \end{cases}$$

The integration in (1.23) can be estimated through

$$G_N = \frac{1}{N} \sum_{i=1}^{N} g(X_i, Y_i) = \frac{\Pi}{4} \tag{1.24}$$

Algorithm 3 illustrates the serial pseudo-code for the Π estimation using Monte Carlo simulations. The number of random points inside the unit circle allow us to calculate the value of Π with an arbitrary precision. The more points generated the better the accuracy for Π. Evidently, such Monte Carlo algorithm in Algorithm 3 is readily adaptable to a parallel environment relying on a Master/Worker pattern with any of the task distribution mechanisms mentioned in the previous section. However, to simplify the example, the static uniform distribution will be considered here. Giving P parallel workers (excluding the master), each worker will receive a task of $\frac{nbMC}{P}$ random simulations to compute its own partial mean. Once finished, these workers can send their results to a leader processor for merging them and computing the final result.

According to Amdahl's Law, we have

- T_1 = execution time for a single worker for all tasks

Algorithm 3 Serial Monte Carlo Simulations for Pi estimation

```
Require: number of Monte Carlo simulations $nbMC$
Ensure: $\Pi$
1: $SumG = 0$
2: for $i = 1$ to $nbMC$ do
3:    Get two random numbers $x_i$ and $y_i \in [0, 1[$
4:    if $(x_i^2 + y_i^2) \leq 1$ then
5:       $SumG = SumG + 1$
6:    end if
7: end for
8: $\Pi \simeq 4 \times \frac{SumG}{nbMC}$
```

- T = execution time for P workers in parallel for all tasks
- T_0 = time for communications (e.g. initialization, task distribution and results collection time, etc)

then the speedup is defined as:

$$\text{Speedup} = \frac{T_1}{T + T_0} \tag{1.25}$$

By considering T is approximately T_1/P which is true for distributed MC applications, there is another important factor which has to be considered, the efficiency. Efficiency is the speedup, divided by the number of workers used and is defined as follow:

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \tag{1.26}$$

1.2.3 Parallel Random Number Generation

Through the overview of numerical methods for computational finance in first section of this chapter, it is clear that Monte Carlo based methods are the most versatile, widely used numerical methods for solving multidimensional problems. However, the convergence rate of Monte Carlo based methods can be very slow compared with other methods. That is why much of the effort in the development of Monte Carlo methods is to speed up the convergence and parallel computing is a promising solution to accelerate the MC simulation speed.

However, a problem in performing MC simulation in parallel computing is the generation of random numbers on parallel processors. Unfortunately, this generation is usually worse than in the serial case. In MC simulation, we need a random number sequence which is independent and without any correlation between numbers. However, in parallel computing each machine has to generate itself a sequence of random numbers and it might be that these sequences could be not independent and correlated to each others (e.g. inter-processor correlation). Such a situation would not appear in the serial case. A good parallel pseudo-random number generator (parallel PRNG) must be able to generate multi sequences of pseudo-random numbers that have the same qualities as serial PRNG; no inter-processor correlation between sequences and those will work for any number of processors. However that still is a difficult problem and is an open research topic. Of course, a bad parallel PRNG is detrimental to MC simulations so choosing a good one is very important. Some researches on parallel PRNGs can be found in Mascagni (1999, 2000) [82, 83] and Coddington and Newell (2004) [35].

In this section, we will present some techniques related to the generation of random numbers on parallel processors. There exists several different techniques to create a parallel PRNG based upon a serial PRNG [82]. We only address here a common technique, the *Sequence Splitting*. Such technique is implemented in PicsouGrid using SSJ package. further details about the SSJ package can be found in Chapter 2.

- Sequence splitting: In this case, the original random number stream is split into blocks and distributed to each processor. Denote the period of the generator is ρ, the number of processors by P and the block length by $L = \frac{\rho}{P}$, we have

$$x_i = x_{pL+i}, p = 1, \ldots, P. \tag{1.27}$$

Then the original stream

$$x_1, \ldots, x_{L-1}, x_L, x_{L+1}, \ldots, x_{2L-1}, x_{2L}, x_{2L+1}, \ldots \tag{1.28}$$

is distributed as follows to processor $1, 2, 3, \ldots$

$$\langle x_1, \ldots, x_L \rangle, \langle x_{L+1}, \ldots, x_{2L-1}, x_{2L} \rangle, \langle x_{2L+1}, \ldots, x_{3L} \rangle, \ldots \tag{1.29}$$

1.3 High Dimensional European Option Pricing and Hedging Validation

It is well known in BS model that there only exists analytical solution for European option pricing in the case of one dimension otherwise for multi-dimension we have to use other approximation approaches such as MC methods. Though the approximated option pricing using MC methods are obtained with a 95% confidence, it is still necessary to find a way to compare them with the analytical results.

In fact, we observed that in some very specific cases we can analytically reduce a basket of assets into a one-dimensional "reduced" asset. The analytical (exact) option price on this "reduced" asset can be computed by using BS formula [19]. Thus in such particular cases, we can validate the approximated results based on the relative error with these analytical results. We will consider both option pricing and hedging problems. Since the reduction of dimension problem has been mentioned in many text books of computational finance but without the details of mathematical proofs, we will present these in the next sections. This problem of dimension reduction will also be used to validate the benchmark suite for evaluation of grid middleware performance in Chapter 4.

1.3.1 Reduction of the dimension in basket option pricing

Consider a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t, t > 0), \mathbb{P})$, equipped with a d–dimensional standard Brownian Motion $W_t = (W_t^1, ..., W_t^d)$. Consider a free risk asset $S_t^0 = e^{rt}$, with fixed interest rate r, and a basket of d assets $S_t = (S_t^1, ..., S_t^d)$, solution of the SDEs (1.17)

1.3.1.1 Payoff function as product of assets

We consider a European option on a basket of d assets S_t, with a payoff function $\Phi(\Sigma_t)$ depending only on the "reduced" variable $\Sigma_t = f(S_t)$, where

$$f(x) = \prod_{i=1}^{d} x_i^{\alpha_i}, \forall x = (x_1, ..., x_d) \tag{1.30}$$

with a given set $\big(\alpha_1, ..., \alpha_d\big) \in \mathbb{R}_+$. We aim to find the SDE satisfied by the "reduced" asset Σ_t in dimension one. To do so, we apply the multi–dimensional Ito Lemma to $f(S_t)$, we get

$$df(S_t) = \sum_{i=1}^{d} \frac{\partial f}{\partial s_i}(S_t) dS_t^i + \frac{1}{2} \sum_{i,j=1}^{d} \frac{\partial^2 f}{\partial s_i \partial s_j}(S_t) S_t^i S_t^j \sum_{k=1}^{d} a_{ik} a_{jk} dt \tag{1.31}$$

With $f(\cdot)$ defined in (1.30), we compute the first term of (1.31),

$$\sum_{i=1}^{d} \frac{\partial f}{\partial s_i}(S_t) dS_t^i = \prod_{i=1}^{d} (S_t^i)^{\alpha_i} \sum_{i=1}^{d} \frac{\alpha_i}{S_t^i} dS_t^i$$

by using the definition of dS_t^i in (1.17), we have

$$\begin{aligned}\sum_{i=1}^d \frac{\partial f}{\partial s_i}(S_t)dS_t^i &= \prod_{i=1}^d (S_t^i)^{\alpha_i} \sum_{i=1}^d \frac{\alpha_i}{S_t^i} S_t^i \left(rdt + \sum_{k=1}^d a_{ik} dW_t^k \right) \\ &= \prod_{i=1}^d (S_t^i)^{\alpha_i} \sum_{i=1}^d \alpha_i (rdt + \sum_{k=1}^d a_{ik} dW_t^k)\end{aligned}$$

We compute the second term of (1.31), we get

$$\prod_{i=1}^d (S_t^i)^{\alpha_i} \sum_{i,j=1}^d \left[\left((\frac{\alpha_i \alpha_j}{S_t^i S_t^j})_{i \neq j} + (\frac{\alpha_i(\alpha_i - 1}{S_t^i S_t^j}))_{i=j} \right) S_t^i S_t^j \sum_{k=1}^d a_{ik} a_{jk} \right] dt$$

Hence, by identifying $\Sigma_t = f(S_t) = \prod_{i=1}^d (S_t^i)^{\alpha_i}$ then Equation (1.31) becomes:

$$\begin{aligned}\frac{d\Sigma_t}{\Sigma_t} &= \left(\sum_{i=1}^d \alpha_i r + \frac{1}{2} \sum_{i,j=1}^d \left((\alpha_i \alpha_j)_{i \neq j} + (\alpha_i(\alpha_i - 1))_{i=j} \right) \sum_{k=1}^d a_{ik} a_{jk} \right) dt \\ &+ \sum_{i,k=1}^d \alpha_i a_{ik} dW_t^k\end{aligned} \tag{1.32}$$

Considering the process X_t defined by $X_t = \sum_{i,k=1}^d \alpha_i a_{ik} W_t^k$, then Equation (1.32) reduces to

$$\frac{d\Sigma_t}{\Sigma_t} = \underbrace{(r - \widehat{\delta})}_{\widehat{\mu}} dt + dX_t \tag{1.33}$$

where

$$\widehat{\delta} = r - \left(\sum_{i=1}^d \alpha_i r + \frac{1}{2} \sum_{i,j=1}^d \left((\alpha_i \alpha_j)_{i \neq j} + (\alpha_i(\alpha_i - 1))_{i=j} \right) \sum_{k=1}^q a_{ik} a_{jk} \right)$$

could be viewed as the dividend yield by the "reduced" asset Σ.

1.3.1.2 The particular case of Geometric Average of d assets

We consider the particular case $\alpha_i = \frac{1}{d}$, $i = 1, \ldots, d$ and $S^i, i = 1, \ldots, d$ are independent assets. This means that $a_{ik} = 0$, for $i \neq j$ and for a given $\sigma > 0$ we set $a_{ii} = \sigma$, $\forall i = 1, \ldots d$. Now we get

$$f(x) = \prod_{i=1}^d x_i^{\frac{1}{d}} \tag{1.34}$$

Start from (1.33)

$$\frac{d\Sigma_t}{\Sigma_t} = \underbrace{(r-\widehat{\delta})}_{\widehat{\mu}}\,dt + dXt$$

with now

$$\widehat{\delta} = (\frac{\sigma^2}{2} - \frac{\sigma^2}{2d})$$

and

$$X_t = \sum_{i=1}^{d}\sum_{k=1}^{d} \alpha_i a_{ik} W_t^k = \sum_{i=1}^{d} \frac{1}{d} a_{ii} W_t^i = \frac{1}{d}\sigma \sum_{i=1}^{d} W_t^i.$$

Define $Z_t = \frac{1}{\sqrt{d}}\sum_{i=1}^{d} W_t^i$. Then standard computations show that $\mathbb{E}\big[Z_t Z_s\big] = t \wedge s$ which implies Z_t as a standard BM on the probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t, t > 0), \mathbb{P})$. Finally, the equation (1.3.1.2) becomes,

$$\frac{d\Sigma_t}{\Sigma_t} = \left(r + \frac{\sigma^2}{2d} - \frac{\sigma^2}{2}\right)dt + \frac{\sigma}{\sqrt{d}} dZ_t \tag{1.35}$$

The asset Σ_t is said to follow a geometric Brownian Motion. By applying the Ito Lemma for the function $F(\Sigma_t) = \log(\Sigma_t)$ and assuming that $\Sigma_t > 0, \forall t$, we have

$$d\log\Sigma_t = \left(\underbrace{r + \frac{\sigma^2}{2d} - \frac{\sigma^2}{2}}_{\widetilde{\mu}} - \underbrace{\frac{\sigma^2}{2d}}_{\frac{1}{2}\widetilde{\sigma}^2}\right)dt + \underbrace{\frac{\sigma}{\sqrt{d}}}_{\widetilde{\sigma}} dB_t$$

$$\log\Sigma_t = \log\Sigma_0 + \big(\widetilde{\mu} - \frac{\widetilde{\sigma}^2}{2}\big)t + \widetilde{\sigma}B_t$$

which leads to the explicit solution

$$\Sigma_T^{\Sigma_t,t} = \Sigma_t \exp\Big(\big(\widetilde{\mu} - \frac{\widetilde{\sigma}^2}{2}\big)(T-t) + \widetilde{\sigma}B_T\Big), \forall t \in [0,T]. \tag{1.36}$$

1.3.2 Option price formula for one-dimensional BS European option

We recall shortly some basic explicit formulas of financial engineering. Consider a call European option on the asset Σ_t modelled by Equation (1.36). We can compute such option value by using the Black-Scholes formula [19, 86]. The call option value at time t is,

$$\widetilde{V}(\Sigma_t, t) = \mathbb{E}\big[\Phi(\Sigma_T^{\Sigma_t,t})\big]$$

with $\Phi(x) = (x-K)^+$. Then a simple computation leads to

$$\widetilde{V}(\Sigma_t, t) = \Sigma_t N(d_1) - K\exp(-r(T-t))N(d_2) \tag{1.37}$$

where $d_1 = \dfrac{\log\left(\frac{\Sigma_t}{K}\right) + (\widetilde{\mu} + \frac{1}{2}\widetilde{\sigma}^2)(T-t)}{\widetilde{\sigma}\sqrt{(T-t)}}$, $d_2 = d_1 - \widetilde{\sigma}\sqrt{(T-t)}$ and $N(\cdot)$ is the cumulative distribution function of the Gaussian law N(0,1),

$$N(d_1) = \int_{-\infty}^{d_1} \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} du$$

- The delta $\widetilde{\Delta}_t$ of the option price $\widetilde{V}$ is defined by

$$\widetilde{\Delta}_t = \frac{\partial \widetilde{V}(\Sigma_t, t)}{\partial \Sigma_t} = N(d_1) \tag{1.38}$$

- The gamma $\widetilde{\Gamma}_t$ of the option price $\widetilde{V}$ is defined by

$$\widetilde{\Gamma}_t = \frac{d^2\widetilde{V}(\Sigma_t, t)}{\partial \Sigma_t^2} = \frac{\partial \widetilde{\Delta}}{d\Sigma_t} \tag{1.39}$$

- Denote $\tau = T-t$ time to maturity. The theta $\widetilde{\Theta}_t$ of the option price $\widetilde{V}$ is defined by

$$\widetilde{\Theta}_t = -\frac{\partial \widetilde{V}(\Sigma_t, t)}{\partial \tau} = -\Sigma_t \frac{\sigma}{2\sqrt{\pi}} N(d_1) - rKe^{-r\tau}N(d_2) \tag{1.40}$$

- The rho $\widetilde{\rho}_t$ of the option price $\widetilde{V}$ is defined by

$$\widetilde{\rho}_t = \frac{\partial \widetilde{V}(\Sigma_t, t)}{\partial r} = (T-t)Ke^{-r(T-t)}N(d_2) \tag{1.41}$$

1.3.3 Option price formula for basket option based on the reduction technique

We aim to compute the price $V(S_0, 0)$ of a call European Geometric Average option on the basket of d assets, where S_0 implies the basket of assets price at initial time of the contract. The pseudo–code of such option pricing using MC methods was described in Algorithm 1. However, this call option price $V(S_0, 0)$ with payoff function $\Phi(x) = (x-K)^+$ is also given by $\widetilde{V}$ in (1.37), where $\Sigma_t = f(S_t) = \prod_{i=1}^d S_t^{i\,\frac{1}{d}}$.

- The vector of Deltas $\left(\Delta_t^i, i = 1, \dots, d\right)$ is the first order derivative with respect to the change of the underlying asset prices $\left(S_t^1, \dots, S_t^d\right)$ of the basket option price $V(S_0, 0)$

is computed as follows,

$$\Delta_t^i = \frac{\partial}{\partial S_t^i}\big(V(S_0,0)\big) = \frac{\partial}{\partial S_t^i}\big(\widetilde{V}(\Sigma_t,t)\big) = \frac{\partial}{\partial S_t^i}\Sigma_t \times \frac{\partial}{\partial \Sigma_t}\widetilde{V}(\Sigma_t,t) = \frac{\partial}{\partial S_t^i} f(S_t) \times \widetilde{\Delta}_t \tag{1.42}$$

where $\widetilde{\Delta}_t$ is given in (1.38). By replacing $f(S_t)$ in (1.34) we get

$$\prod_{i=1}^{d} \Delta^i = \Big(\frac{1}{d}\widetilde{\Delta}\Big)^d \frac{1}{\prod_{i=1}^{d} S_t^i} \left(\sqrt[d]{\prod_{i=1}^{d} S_t^i}\right)^d = \Big(\frac{1}{d}\widetilde{\Delta}\Big)^d. \tag{1.43}$$

- Gamma is the second derivative with respect to the change in the underlying prices $\big(S_t^1,\dots,S_t^d\big)$ of the basket option price $V(S_0,0)$. Therefore the matrix of Gamma $\big(\Gamma_t^{ij}, i,j = 1,\dots,d\big)$ is computed as follows :

$$\begin{aligned}\Gamma_t^{ij} &= \frac{\partial^2}{\partial S_t^i S_t^j}\big(V(S_0,0)\big),\ i,j = 1,\dots,d \\ &= \frac{\partial^2}{\partial S_t^i S_t^j}\Sigma_t \widetilde{\Delta}_t + \frac{\partial}{\partial S_t^i}\Sigma_t \frac{\partial}{\partial S_t^j}\Sigma_t \widetilde{\Gamma}_t\end{aligned} \tag{1.44}$$

where $\widetilde{\Delta}_t$ and $\widetilde{\Gamma}_t$ are given respectively in (1.38) and (1.39).

- Theta Θ hedging for the option price V

$$\Theta_t = \frac{\partial}{\partial \tau}\big(V(S_0,0)\big) = \frac{\partial}{\partial \tau}\widetilde{V}(\Sigma_t,t) = \widetilde{\Theta}_t, \text{ given in (1.40)} \tag{1.45}$$

- Rho ρ hedging for the option price V

$$\rho_t = \frac{\partial}{\partial r}\big(V(S_0,0)\big) = \frac{\partial}{\partial r}\widetilde{V}(\Sigma_t,t) = \widetilde{\rho}_t, \text{ given in (1.41)} \tag{1.46}$$

1.4 Conclusions

In the modern derivative market, option contracts have been particularly very actively traded products with very large trading volume. In the current financial crisis, in order to adapt with more and more complex demands from the derivative market, many sophisticated option contracts (e.g. with portfolios of thousand of underlying assets) have been issued hence bringing about complicated models and computing methods. Therefore the mathematical models and numerical methods of option pricing and hedging have been developed for more or less 30 years, thus their evolution is not over due to the endless demanding of the market. Moreover, within a time critical market, these contracts usually have to be priced and traded within a short delay and often are at the upper limit of the available computing resources. Obviously, that requires more addition of manpower and computing power to handle all the models and methods before being able to obtain the results. Therefore applying parallel computing (aka. high performance computing) becomes a must to address these time constraints.

It is well-known that the power of parallel computing reduced computational time for large problems by splitting them into smaller ones and solving them concurrently. Recently, as a sub-domain of parallel computing, grid computing, which has been developed for the last decade, is a promising solution for such challenges in finance. Although grid computing includes many advantages, mentioned in this section, it also has some difficulties due to its heterogeneity, distributed implementation, application deployment, etc. Hence to profit from grid computing, especially the finance industry, one needs a grid-enabled framework that should be able to abstract the financial "users" from the underlying resources, thus providing a uniform, scalable, and robust computing environment. The "users" here include the common users who want to run their applications faster on a grid and also the financial software engineers who want to focus on the pricing algorithm and not worry much about grid technical issues. In the next Chapter, we are going to introduce such a framework, named PicsouGrid, which was created to address these issues.

Chapter 2

PICSOUGRID : A GRID FRAMEWORK FOR FINANCIAL DERIVATIVE PRICING APPLICATIONS

This chapter introduces the design and implementation of a grid framework for financial derivative pricing applications, named PicsouGrid. PicsouGrid was first presented in the 2^{nd} e-Science and Grid Computing conference in 2006, see [18] and later in the 2007 International Symposium on Grid Computing, see [110]. This material is also part of the "Risk 1 sub-project" of the ANR GCPMF project [1]

[1]http://www.pricingpartners.com/anr-cigc-gcpmf/

2.1 Introduction

We introduce an open source distributed computing framework in this financial context which specially targets derivative pricing with the capability to provide fault tolerance, load balancing, dynamic task distribution and deployment in a heterogeneous environment (e.g. grid computing). Such a framework should be able to abstract the user and application interface from the underlying resources, so as to provide a uniform, scalable, and robust computing environment. This framework should have the capability to help the application developers who want to focus on the pricing algorithm and not worry about grid technical issues or the complexity of coordinating multi-threaded, distributed and parallel programs. In addition to serving the practical goal of harnessing idle processor time in a grid environment, such a framework proved to be a valuable experimental workbench for studying computational finance particularly in high dimensional problems.

2.2 Related Work

2.2.1 General Purposes Grid Programming Frameworks

In Zenios 1999 [128], the author mentioned the use of high performance computing in finance for the last ten years of the 20^{th} century and discussed future needs. The research work presented in [128] focused upon three major sections that relate respectively to derivative instruments pricing (i.e. option pricing), integrated financial product management and financial innovation. For each field, the author briefly described some basic model, the computational issues and pointed out the extensions of the models that were facilitated through the use of high performance computing. However, the paper only addressed in details the technologies of how to apply high performance computing to each individual field.

There also exist many distributed frameworks based on a Master/Worker architecture which aim to be general purpose, either in a global computing or peer-to-peer environment [7, 6] or in a grid context [8, 33, 122]. For example, OurGrid [8] including MyGrid [33] is a complete solution for running Bag-of-Tasks (BoT) applications on computational grids. In fact, BoT applications are the parallel applications whose tasks are independent of each other and do not require any synchronization between them. Such applications can easily be distributed through a Master/Worker underlying architecture. Meanwhile, the Satin [122] is a system which has been introduced for running divide and conquer programs on distributed memory systems. Divide and conquer is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Further details of these approaches will be discussed in the following sections.

OurGrid project OurGrid [8] relies upon a peer-to-peer network of resources owned by a community of grid users. It works based on assumptions that it does not need quality of service guarantees and that there are at least two peers in the system willing to share their resource in order to have access to more resources. Hence by adding resources to the peer-to-peer network and sharing them with the community, a user gains access to all the available resources on it. All the resources are shared respecting each provider's policies and OurGrid strives to promote equity in this sharing. The OurGrid middleware is written in Java, hence easy allowing any application capable of running a JVM to be utilized on a grid. The goal of OurGrid is to be fast, simple, scalable and secure. In order to archive such goals and avoid others shortcomings, OurGrid reduced its scope to support only BoT based applications. In this particular scope, OurGrid [8] delivered a fast execution of applications without demanding any quality of service (QoS) guarantees from the infrastructure. It also provides a secure environment. Finally, OurGrid provides a fault tolerance mechanism by which it will replicate tasks in multiple resources or re-submit tasks that recently failed.

A series of success stories of using OurGrid in e-sciences provides insights on how OurGrid is helping people to complete their computations. For example, OurGrid has been used to provide a computational grid infrastructure for a system that supports sustainable water resources management. However, OurGrid has not been applied yet to the financial domain.

As part of the OurGrid project, MyGrid [33] is the scheduling component of the OurGrid solution. MyGrid focuses on the complexities involved in using grid technology and the slow deployment of existing grid infrastructure that can potentially bring to bear for use of grid for BoT applications. Hence MyGrid aims to easily enable the execution of BoT application on whatever available resources. Its goal is to provide a global execution environment composed of all processors that the user has access to. Hence, the least user gets involved into grid aspects, the better it is. Towards this goal, MyGrid worked on minimizing the installation effort by provioing two interfaces : the GridMachine (GuM) and the GridMachineProvider (GuMP) interfaces. The MyGrid scheduler, which is responsible for starting and monitoring the tasks, uses GuM as an abstraction for a processor and GuMP as an abstraction for a set of processors. The first interface contains the minimal set of services that must be available in a processor (to allow its use as part of a grid) including remote execution, tasks cancellation, file transfer mechanism and connectivity check. That set helps MyGrid work without knowing details about the grid processors. This also eases the addition of new resources to MyGrid, since implementing the interface is all that is needed. Using GuM interface, user will be provided with methods to communicate with grid machines. MyGrid has three native implementations of GuM: (i) UserAgent, (ii) GridScript, and (iii) GlobusGridMachine. UserAgent (UA) is simple Java-based implementation of GuM designed for the situation on which it is easy for the user to install software on the grid machines. UA is extended to implement other methods such as instrumentation and fault tolerance. The communication between MyGrid and UA is made using Remote Method Invocation (RMI). GridScript (GS) provided the same role as UA but using scripts instead of using a Java process. The last one, GlobusGridMachine (GGM) aims to interoperate with Globus 3.0[2]. The second interface, GuMP, is used in case of using MyGrid for parallel supercomputing, therefore it will not be discussed here. MyGrid is designed to make grid computing easier for the users of Bag-of-Tasks typed e-science applications. It was successfully used for applications developed to support scientific problem, however none of them was specified in financial sector.

Satin project : More recently, Satin [122] is a divide and conquer system based on Java and has been designed for distributed memory machines. The goal of Satin is to support the execution of distributed applications on hierarchical wide-area clusters (e.g., the DAS [13]). The reason is that the divide and conquer model will map efficiently on such systems, as the model is also hierarchical. Satin is designed as a compiler-based system in order to achieve high performance. Satin is based on the Manta [79] native compiler, which supports highly efficient serialization and communication. Satin project provides a programming model which is an extension of the single threaded Java model to ease the application development. In Satin's programming model, the authors introduce three new keywords to Java language, *spawn*, *sync* and *satin*. The *spawn* keyword is used in front of a method invocation. The parallelism is achieved in Satin by running different spawned method invocations on different machines. When *spawn* is placed in front of a method invocation, conceptually a new thread is started which will run the method. (The implementation of Satin, however, eliminates thread creation altogether.) The spawned method will run concurrently with the method that executed the spawn. In Satin, spawned methods always run to completion. The *sync* operation waits until all spawned calls in this method invocation are finished. The return values of spawned method invocations are undefined until a *sync* is reached. The *satin* modifier must be placed in front of a method declaration, if this method is ever to be spawned.

Through such frameworks, users are able to manage grid resources and deploy, customize

[2]www.globus.org/

or develop their own grid-enabled applications on grid. As we briefly highlighted above, OurGrid [8], MyGrid [33] and Satin [122] projects provided complete solutions for running applications on computational grids. While OurGrid and MyGrid addressed the scope of Batch-Of-Task applications which is well suited with grid environment, the Satin project is a system for running divide and conquer program for distributed memory machines.

Each project has been followed by a series of success stories in e-science applications however pricing financial applications (e.g. option pricing) have not been addressed. In the next section, we are going to introduce grid solutions which are dedicated to financial and business problems.

2.2.2 Use of High Performance Computing in Finance

2.2.2.1 Academic Centric Project

There have been several research projects that address HPC for finance and computational economies such as [93, 125, 59]. Recently, the AURORA "Advanced Models, Applications and Software Systems for High Performance Computing" [3] funded by the Austrian Science Fund (FWF), focused on high level software for HPC systems, with related research issues ranging from models, applications and algorithms to languages, compilers and programming environments. During the last few years, the AURORA projects research work has shifted to grid computing. Thus, the core of the AURORA project is actively involved in various grid projects and provides an ideal infrastructure for application groups to grid-enable applications. It is being able to instantiate large real-world models and solves problems of unprecedented size and complexity.

The AURORA sub-project High Performance Computing in Finance (AURORA Financial Management System (FMS)) [93] is a part of the AURORA project. The AURORA FMS addressed the area of computational finance, especially in modelling and optimization of large stochastic financial management models and in pricing financial products. The AURORA FMS provided a modular decision support tool for portfolio and asset management. It is presented as a component-based computational financial management system to solve large scale investment problems. The core of the system is a linear or convex program, which due to its size and structure is well suited for parallel optimization methods. The system also contains pricing models for finance such as future, option contracts etc. However, the goal of the AURORA FMS is only to ease the development of high performance applications for finance and to provide the development software tools to support it. It does not support the execution of these the applications on the grid. In order to cover this lack of

[3]http://www.univie.ac.at/sor/aurora6/

grid technology supports, further works [125, 59] have provided a distributed environment and execution tools for the AURORA FMS.

In [125] the authors present a complex problem modelling and solving environment for large scale computational finance modelling systems which grew out of the extension of the AURORA Financial Management System. The implementation of the system follows the Open Grid Service Environment (OGSE) Service Stack. This environment has been used to model large-scale computational finance problems as abstract workflows with meta-components and instantiate such workflows with different components based on semantic matching. The authors outline the nature of large-scale financial problems in general and give an example for a typical problem in this area (for example a computationally demanding task such as end of day settlement processing). Furthermore, they used this example to show how to use a grid environment to enhance performance by exploiting intra-component and workflow parallelisms.

Work in [59] continues to present some examples from the field of computational finance, which substantiate the need for grid technologies and take a closer look at the implementation issues, applying the concepts of enterprise service business for parallel process orchestration including portfolio management and multi stage stochastic asset liability management. The first example was introduced in the early 1950s by Markowitz [81] which requires solving a complex iterative process involving the execution of many “heterogeneous” algorithms. Such algorithms can be spread over cluster or other similar architectures (e.g. multi-core processors). The second example consists of valuation a single pension fund contract for insurance companies. If the company is aiming at solving and optimizing an enterprise-wide investment strategy, every contract has to be calculated. For a large insurance company, tens of thousands of contracts have to be optimized (many “heterogeneous” scenarios), while communication is still necessary in order to summarize the results for generating the overall investment strategy. These contracts calculations can be conducted in parallel to a large extent. For both examples, the implementation applying the Parallel Process Workflow Variation pattern [121] is quite suitable from a computer science point of view.

More recently, BeInGrid, an European funded research project, showed a numerous of success stories from the use of grid solutions by the businesses parties involved in the project. The use of grid computing in financial applications is part of such series of success stories. Each story is a specific case study in financial sector that is parameterized in order to be adapted within the grid environment including enhancing the system performance, managing application services, computing large data sets, calculating risks, etc. The solutions include:

- using commercial HPC technology for financial needs (e.g. Shinhan Bank used Oracle

10g to build a robust, secure platform for its critical business system, UniCredit Group employed DataSynapse GridServer application virtualization software that enabled application services to be distributed and executed across a shared, heterogeneous infrastructure.);

- building a dedicated grid infrastructure for finance (e.g. Merrill Lynch has developed an enterprise computing grid that allows it to run simulations and risk analysis for high value derivatives trades faster);

- parallelizing the computational algorithms (e.g. Axa Life Europe Hedging Services Ltd used some parallel pricing algorithm to determine the price of the guarantee [92])

BeInGrid is an evident example proving the successful collaboration between academic institutions and financial partners in the domain of grid computing for finance. Many successful case studies of partners involved in the project have been presented though its series of success stories. Each partner proposed his own particular case study and then finding a specific grid solution to solve its problems. It is the fact that the goal of this project is not to provide a generic methodology for the use of grid computing in finance. That can be easy to understand due to a very large number and the heterogeneity of scenarios in financial domain and therefore it is difficult to figure out a common solution that may cover even some of such scenarios.

During the last recent years, the interest and the strong involvement of academic scholars in the domain of grid computing for finance have arisen. Many workshops whose scope addressed the HPC for finance domain have been organized (e.g. workshops in conjunction with IPDPS[4], Supercomputing[5] conferences). The main goal of the workshops was to bring together financial institutions, banking practitioners and researchers from the complementary fields of high performance computing and computational finance.

2.2.2.2 Financial Industry Solutions

In contrast with the academic sector, as an article in grid Today [57] argued that the financial services sector will be a driving force in the application of grid technologies and the competition between HPC technology companies in financial market is very effervescent. Especially the recent additional set of regulatory constraints for the financial industry, like

[4]http://www.cs.umanitoba.ca/ pdcof/

[5]http://research.ihost.com/whpcf/

Basle II, forces large financial companies, like BNP Paribas, to create their own grid applications. Therefore the leading technology companies in providing HPC solutions using multi-core or grid computing for general purposes such as Datasynapse [6], Platform Computing [7], Digipede [8], etc; have been constantly concerned with financial services for the last few years.

As an example, DataSynapse [111] a leading company in dynamic application service management software for data centre has built a grid platform named GridServer. GridSever is application infrastructure software for general purposes that virtualizes and distributes application services and executes them in a guaranteed, scalable manner over existing computing resources. Client applications submit requests to the grid environment and GridServer dynamically provisions services to respond to the request. Multiple client applications can submit multiple requests in parallel and GridServer can dynamically create multiple service instances to handle requests in parallel on different grid nodes. GridServer is best suited to applications that can be divided into independent tasks (e.g. BoT applications). Most of such applications consist of overnight batches applications which exhibit a large volume of trading transactions. Although having the name related with grid computing, GridServer has been preferably installed on multi-core processors platform (e.g. BNP Paribas London has deployed GridServer across Hewlett Packard blades).

Another example is Platform Computing [36] which provides a distributed computing framework, named Platform Symphony, that makes it practical to develop and manage distributed computing services in a coherent way. The Platform Symphony architecture has three distinct components including a software development kit (SDK) for application development, workload management, a computing resource management module and a management console for virtual control. Application developers have to adapt their business application to Platform Symphony through the given SDK at client side. Then at service side, they have to create a service container which has the responsibility to perform the tasks triggered by the client side. Once such services have been created, developers can deploy them in a grid environment by using the computing resource management module then the system itself will take care of the rest.

As similar as Platform Computing, Digipede [115] provides both a platform for distributed computing and a framework for distributed programming. The Digipede network platform has 4 components including an agent component which manage each of individual desktop, server, cluster nodes and tasks that run on such resource; a server component

[6]www.datasynapse.com

[7]www.platform.com

[8]www.digipede.net

manages the workflow through the system; a workbench component lets users define and run jobs; and a control component provides a browser based tool for system administration.

Rapidmind [90] and ClearSpeed [55] are the technology companies focusing on multi-core processors architecture. They provide software development tools for software parallelism and performance for general purposes. Parallelizing financial applications is only a part of their case studies.

Due to the common large scope that the research projects and technology companies have targeted in the domain of general purpose high performance computing, none of them aim to provide a dedicated HPC solution for finance. Those solutions in general try to focus into managing the flow of work, the workload of the system, the latency or the capacity of managing as much as possible of computing resources. All of them are well suited with workflow based applications, with overnight computing applications with a very large volume of computations which however is easy to be divided into many independent smaller volumes, or with simulation based applications which require lots of different scenarios (e.g. Value At Risk (VaR) using Monte Carlo methods) etc. These applications have a common particular characteristic that they only consist of embarrassingly parallel applications (e.g. VaR, European option pricing using Monte Carlo methods) which are easy handled by using the Master/Worker architecture. In case of non-trivial parallel applications (e.g. American option pricing) which require much more communications and synchronization at different phases, no further discussion is mentioned.

Moreover, financial applications are only a small part of their cases study. Although the overall objective is to cover any heterogeneous computing environment, most of the industrial HPC solutions for finance in fact are preferably installed on a stable multi-core processors clustered infrastructure rather than on a true grid. The reasons are simply that grid aspects such as fault tolerance, load balancing, dynamic task distribution and deployment on a heterogeneous environment still cause lot of challenging research and technical issues.

In order to handle the combined issues between business demands and grid technologies needs, we introduce a distributed framework for financial purpose, named PicsouGrid [18]. At a high level of description, PicsouGrid based on a Master/Worker architecture is organized as a platform toward to financial user and application developers. Typically, we assume that financial users are non technical persons. For example, they could be traders who only wish to purchase computation time in order to perform some computation and gain some profits from such trade-off. On the other hand, application developers are financial software engineers who wish to easy implement new financial algorithms that want to profit the unused, otherwise-wasted processing time on their computational resources. In order to satisfy both requirements, PicsouGrid was designed and implemented as an open, market based computational framework. It provides an automatic mechanism to utilize idle computational resources in a distributed and heterogeneous environment. It is supposed to support parallelizing concurrent applications such as Monte Carlo simulations as prototypical applications. To this aim, PicsouGrid provides a dynamic task distribution mechanism. In order to facilitate the deployment of application on a grid environment, PicsouGrid has an abstract away interface with the underlying execution platform. On the application development perspective, starting at the point of building a dedicated distributed framework for finance, we are interested in investigating the financial application requirements, some particular computational finance algorithms, specific numerical methods for finance etc in order to figure out the parallel portions that can benefit from HPC. Hence PicsouGrid provides a flexible mechanism to develop any distributed computational finance algorithm (i.e. especially the Monte Carlo based algorithm for derivative pricing) including the embarrassingly/in-trivial parallel applications. The overall goal is to develop and deploy any parallel financial application (Monte Carlo based) with as less as possible of modifications of their serial implementations.

In Secton 2.3, we will introduce the PicsouGrid framework architecture and present in detail guide lines for application developers.

2.3 PicsouGrid Framework

2.3.1 Motivation

While almost all publications mentioned in section 2.2 cite market time constraints as a key motivation for the parallelization of Monte Carlo-based option pricing algorithms, none explores of efficient, dynamic load balancing, fault tolerance, and deployment upon large systems issues which are key goals we wanted to address through the PicsouGrid framework.

Load Balancing One of the main goals of PicsouGrid framework is to support parallelization the applications (e.g. Monte Carlo based applications). The reason is that many risk analysis (including option pricing) are based on MC simulations, and need to run at least a fixed large enough number of simulations $nbMC$ to achieve the desired accuracy. In a parallel environment this suggests that the simulations are conducted concurrently by the available parallel workers. In a traditional homogeneous parallel environment there is an implicit assumption that these simulations all take the same amount of time, so the division of the simulations is uniform, and typically $nbMC$ total number of simulations are divided by P available workers at each stage, and each worker handles $\frac{nbMC}{P}$ simulations. This solution generates the minimum amount of communications, but slows down the overall computation when the processors are heterogeneous. In fact, in a grid environment it is not possible to assume the processors are uniform. Thus once considering heterogeneousness of the machines and possible failures or slowing down, to achieve these stochastic computations as fast as possible, another solution consists of sending tasks to workers until the master collected enough of simulations results then, useless tasks still running on workers are cancelled. During computation some processors may fail or proceed very slowly, in which case the remaining processors should be able to adjust their workload appropriately. This suggests applying a factor F to the number of processors P, such that $nbMC$ simulations are divided by $F \times P$. In a uniform environment, each processor would handle F packets of size $\frac{nbMC}{F \times P}$, however in a heterogeneous environment the faster processors would acquire $> F$ packets, and the slower processors $< F$, thus providing load balancing. The selection of packet size has an impact on the waiting time for the last packet to be returned (so smaller packets would be better) versus the communications overhead of many packets (so larger packets would be better). Related to this issue of optimal packet size is the degradation in speed-up when additional processors are added. The strategy of using smaller packets would take advantage of the fastest machines in the grid, but generates more communications that may issue a trade-off which is the classic problem of parallel computing, again related to the CCR: Communications-to-Computation Ratio problem.

Fault Tolerance One of the most attractive features of the grid is the ability for a client application to be able to send out (potentially computationally expensive) jobs to resources on the grid, dynamically located at run-time. However, the nature of grid resources means that many grid applications will be functioning in environments where interaction faults are more likely to occur between disparate grid resources, whilst the dynamic nature of the grid, means enter and leave resources at any time, in many cases outside of the applications control. This means that a grid application must be able to tolerate (and indeed, expect) resource availability to be fluid. The goal of fault tolerance feature is to allow applications to continue and finish their works in case of failure concretely for PicsouGrid. That means to figure out the shutdown worker and replace it by another available one as soon as possible. In term of application, since we support both coarse-grained concurrent applications and applications requiring many unrelated and identified tasks, PicsouGrids fault tolerance support restricts itself to failure detection not trying to re-distribute the missing tasks as soon as possible and assuring that the master collect enough of tasks. In case of concurrent application such as an European option pricing using Monte Carlo methods, its result only considers the total number of simulations. Once a fault happens for example the application losses $\frac{nbMC}{P}$ simulations, then it is enough to create another packet of $\frac{nbMC}{P}$ simulations to replace it. However, in case of applications with related and identified tasks (e.g. American option pricing). Thus if a task disappears, we need to identify the index of such missing task in order to re-compute it. We will discuss this issue further in the next section.

Deployment Deployment of distributed applications on large systems, and especially on grid infrastructures, can be a complex task. Grid users spend a lot of time to prepare, install and configure middleware and application binaries on nodes, and eventually start their applications. The problem is that the deployment process consists of many heterogeneous tasks that have to be orchestrated in a specific correct order. As a consequence, the automation of the deployment process is not easy to solve by hand. To address this problem, we propose PicsouGrid to rely upon the use of a generic deployment mechanism provided by a given grid middleware which allow to automate the deployment process. PicsouGrid users only have to describe the configuration to deploy in a simple natural language instead of programming or scripting how the deployment process is executed. For this thesis, and for the PicsouGrid framework, we choose to rely on the ProActive middleware offered deployment mechanism.

Figure 2.1 briefly presents the modular architecture of PicsouGrid.

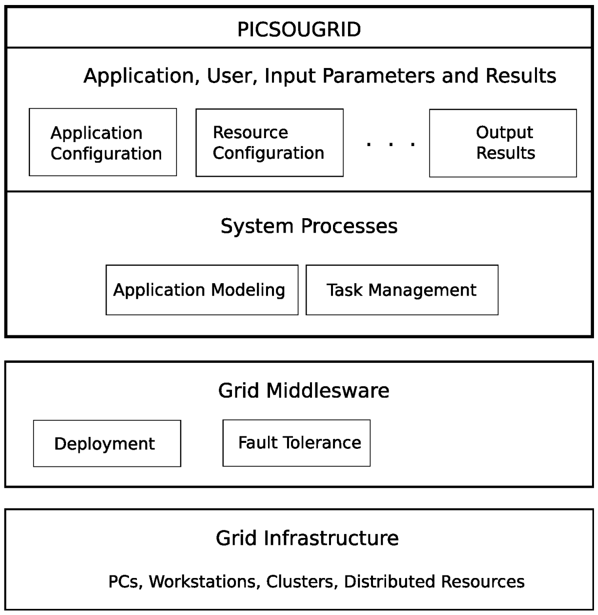

Figure 2.1: PicsouGrid Modular Architecture

2.4 PicsouGrid Architecture

It is well-known that Master/Worker pattern for parallel computing is the easiest and common case for handling distributed computations as simply needs to create a master and a group of workers: tasks have simply to be distributed uniformly to workers and merged later by the master. When the resources are homogeneous, parallelizing financial computations relying on MC simulations along the Master/Worker pattern would be the simplest solution. Basically, MC simulations are independent operations that can naturally be packaged into independent tasks, further allocated to some workers, the results being merged at the master level by applying common operators such as mean, variance, etc (which is typical of European option pricing). So, a Master/Worker based framework dedicated to Monte Carlo based mathematical finance should constitute a useful tool for end-users. Such a framework would hide resource allocation, master and worker computation entities deployment, initialisation and start, etc. We have contributed to the realisation of such a tool relying upon the Master/Worker API and framework available in the ProActive Parallel Suite, getting the so-called ProActive Monte-Carlo API. Details of this API will be discussed further in Chapter 4 as it has been effectively used to run European option pricing computations.

However, the ProActive Monte-Carlo API, and more generally, any simple "naive", that is, single level Master/Worker framework, raises some limitations when used on large-scale heterogeneous infrastructures as Grids: lack of fault tolerance mechanisms to tolerate the loss of a worker or the master, communication overhead from and to the master, static definition of task sizes, and inherent and limited size for the pool of workers (because of technical limitations due to the limit of sockets number to connect the master to its workers). These reasons encouraged us to investigate a more sophisticated framework than the ProActive Monte Carlo API, (e.g. the PicsouGrid framework).

2.4.1 Master, Sub-Master and Worker Hierarchy

Based on the classical Master/Worker architecture, we designed the PicsouGrid framework so that the master acts as an entry point to the system distributed over its own specific nodes (typically groups of computing resources within clusters or a grid). To face poor performance due to the overhead in communication of too many workers and for the extensibility, our PicsouGrid architecture was designed hierarchically as follows: the master controls a set of sub-masters which in turn control a large number of workers, see Figure 2.2. The user accesses the system through the master, which in turn can initiate one or more sub-masters associated with their own workers. With this approach, the number of sub-masters can be increased as will when the number of needed workers becomes too large to be handled by one master.

To ease the programming of parallel financial algorithms we investigated and compared two solutions for communications: the first simply consists in relying on message passing to handle transparently any inter-worker and worker-sub-master communications; the second relies on a shared memory approach (i.e. a shared space of tuples [51] to handle some of these communications). Such a space can be instantiated on demand and is suitable at least when a sub-master and all its workers are deployed on a same cluster with fast communication network. PicsouGrid has been designed and implemented to include both communication paradigms, in order to be a generic architecture supporting the implementation of various parallel algorithms and different programming strategies.

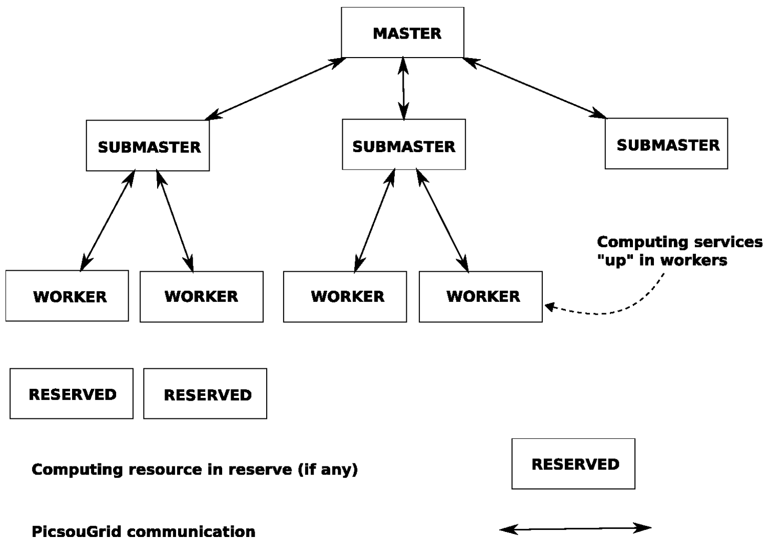

Figure 2.2: Master/Sub-Master/Worker Hierarchy

Figure 2.2 shows the organization of PicsouGrid. The user code is split into classes, which have to inherit from three PicsouGrid classes (master, sub-master 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 (i.e. end-user) application. These Java classes are generics parameterized with the user-defined classes, which of course can have any name and so 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. Finally, the developers of a financial application do not need to care about deployment or fault-tolerance. As long as developers extend the three PicsouGrid classes, PicsouGrid itself deals with those issues.

2.4.1.1 Load Balancing

We implemented a two-featured load balancing mechanism in PicsouGrid that somehow tolerates faults, assuming computing resources can be heterogeneous and unreliable. The first feature is *dynamic* load balancing which 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 (either anonymous or explicitly identified) to process, and asks for a new one when it has finished. This idea is extended to the hierarchical distribution of a set of tasks from the master to the sub-masters, thus ensuring dynamic load balancing of parallel computations on a large number of processors.

This classical strategy has been implemented with both message passing and shared memory communication paradigms:

- In the case of a shared memory paradigm, some initial data and tasks are put in the virtual shared memory by a sub-master. 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 sub-master or read by other workers if required by the algorithm, If needed, the sub-master or some workers can also provide new tasks to be processed in the space.

- Implementing dynamic load balancing with message passing paradigm in PicsouGrid requires to write a little bit more code in the sub-master. The sub-master 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.

The second feature of the load balancing mechanism we need to provide is *aggressive* task distribution. Many risk analysis are based on Monte Carlo simulations, and need to run at least a fixed number of simulations *nbMC* to achieve the required accuracy. To achieve these stochastic computations as fast as possible, considering heterogeneity of the machines and possible failures, a solution consists in sending tasks to workers until the sub-master has collected enough results (not just until having sent enough tasks to the workers). Then, useless tasks which are still running on workers must be cancelled by the sub-master. This strategy takes advantage of the fastest machines in the grid, but generates more communications that may slow down the computation. Another solution consists in sending big tasks to the P workers of the sub-master: 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 more than one processor fail. This solution generates the minimum amount of communications.

2.4.1.2 Fault Recovery

The fault recovery mechanism available in PicsouGrid is an applicative-level one. PicsouGrid can detect faults in a simple way: the server regularly pings its sub-masters and a sub-master 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, sub-masters regularly checkpoint the received results. When a worker disappears, its sub-master first requests a node from the reserve machines, see Figure 2.2. 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 sub-master fails to respond. In that case, the master requests a new node from the reserved pool; if the pool is empty, the master is chosen since it does not perform a lot of computations. A new sub-master is started and the master sends it the interrupted task, and the last check-pointed state from the dead sub-master; each worker from the initial group is re-attached to the new sub-master.

Besides, the applicative-level fault recovery management, it might be the case that the supporting grid technology also offers some “built-in” fault-tolerance mechanism. For instance, the ProActive grid technology is able to checkpoint objects running on the grid, for further transparent recovery [11].

2.4.1.3 Deployment Mechanism

One of the objectives of the PicsouGrid framework is to deploy applications anywhere without changing the source code. Therefore, we see that the creation, registration and discovery of resources have to be done externally to the application. This key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. In order to abstract away the underlying execution platform, and to allow a source-independent deployment, PicsouGrid satisfies the following:

- Having an abstract description of the distributed entities of a parallel program.

- Having an external mapping of those entities to real machines, using actual creation, registry, and lookup protocols.

In fact, PicsouGrid utilizes two deployment descriptors according to the ProActive deployment model [11], separating by the different role for a user of a distributed environment: administrator and application developer. In the distributed environment resources deployment descriptor, we describe:

- The resources provided by the infrastructure, and
- How to acquire the resources provided by the infrastructure.

In the application deployment descriptor we describe:

- How to launch the application,
- The resources needed by the application, and
- Which resources provider to use.

2.4.1.4 Discussion: Multi-phases Master/Worker Computations and Fault Recovery

Master/Worker classical architecture pattern assumes one single and uniform shot of computation: the master is given a list of tasks and has the duty to make them solved by its workers. Use of the Master/Worker pattern for option pricing requires more. Indeed, some option pricing applications such as American option pricing require complex models proceeding in phases, possibly with convergence iterations, where the results from one stage must be gathered from all workers, merged, and some updated parameters calculated and distributed back to each worker. This communication overhead can be a major bottleneck for parallel algorithm implementations, and the impact increases with the number of parallel processors. For example, with the complex algorithms for American option pricing, described in Chapter 1, the distributed approaches for such pricing algorithms are not simply embarrassingly parallel. Figure 2.3 introduces a generic algorithm flowchart for these distributed American option pricings mentioned above. Obviously, it is necessary to do computations in two different parallel phases. The first phase is usually used to characterize/compute the optimal exercise boundary and then the American option prices will be computed in the second phase. The first phase requires a backward iterative cycle upon opportunity dates where all workers must be updated with newly calculated values from the previous iteration, and its partial tasks also require some small iterative cycles to converge. The second phase consists of straightforward Monte Carlo simulations.

In this situation, especially in the first phase, it is quite difficult to synchronize and recover from worker failures at arbitrary points in the overall computation. For example, consider P workers for J different parameterized tasks. At a given iteration i (aka. opportunity date), the master must receive results from J tasks in order to pass to iteration *i-1*. However, due to a very slow worker or a shutdown one, only *J-1* tasks have been completed. The master must wait the missing task results. Trying to apply fault detection and recovery would result in 1) determining the missing worker, i.e. the master will send a ping message to all workers and wait for their answers; 2) Once finding the missing worker, the master would now replace its duties by another worker. During that time, other workers would be in waiting situation until the master receives the last simulation results. Such delay will become a major factor reducing the system performance especially in case of more than one worker missing. Moreover, we can not remove such delay by letting the workers continue their computation without waiting the missing tasks because the computation at iteration *i-1* strongly depends on the calculated results at the previous iteration.

As we noticed and further exploited as explained next, our flexible load-balancing mechanism in PicsouGrid could suffice to resist the occurence of faults, by not trying to detect and recover faulty computation entities, but able to finish the computations bypassing the faulty entities. Moreover, the high frequency appearance of new American option pricing algorthims such as Longstaff and Schwartz (2001) [77], Ibanez and Zapatero (2004) [63] and Picazo (2002) [94] has pressed us to make PicsouGrid applications be able to adapt any new algorithm with less of modification.

Thus, we shifted the focus from applicative fault recovery to more autonomy and flexible distribution of tasks for complex option pricing algorithms. The resulting approach is presented in the next section.

2.4.2 Combined Master/Worker Hierarchy

As mentioned above, using a standard Master/Worker pattern extended hierarchically to Master/Sub-Master/Worker would require the need of a synchronization step when workers return results to sub-masters and from sub-masters to master. In order to offer fault-resilience while avoiding so strong synchronizations, we investigated an alternative architecture for the PicsouGrid framework where masters and workers are merged into general abstract simulators. Still, applications should be designed along the Master/Worker pattern: workers compute, master distribute work and merge the results.

2.4.2.1 New architecture

We merged both master and worker patterns into an abstract simulator module. Consequently, such simulator module (set of classes) includes the master's methods such as merging methods, and also the partial calculation methods of workers. One advantage is

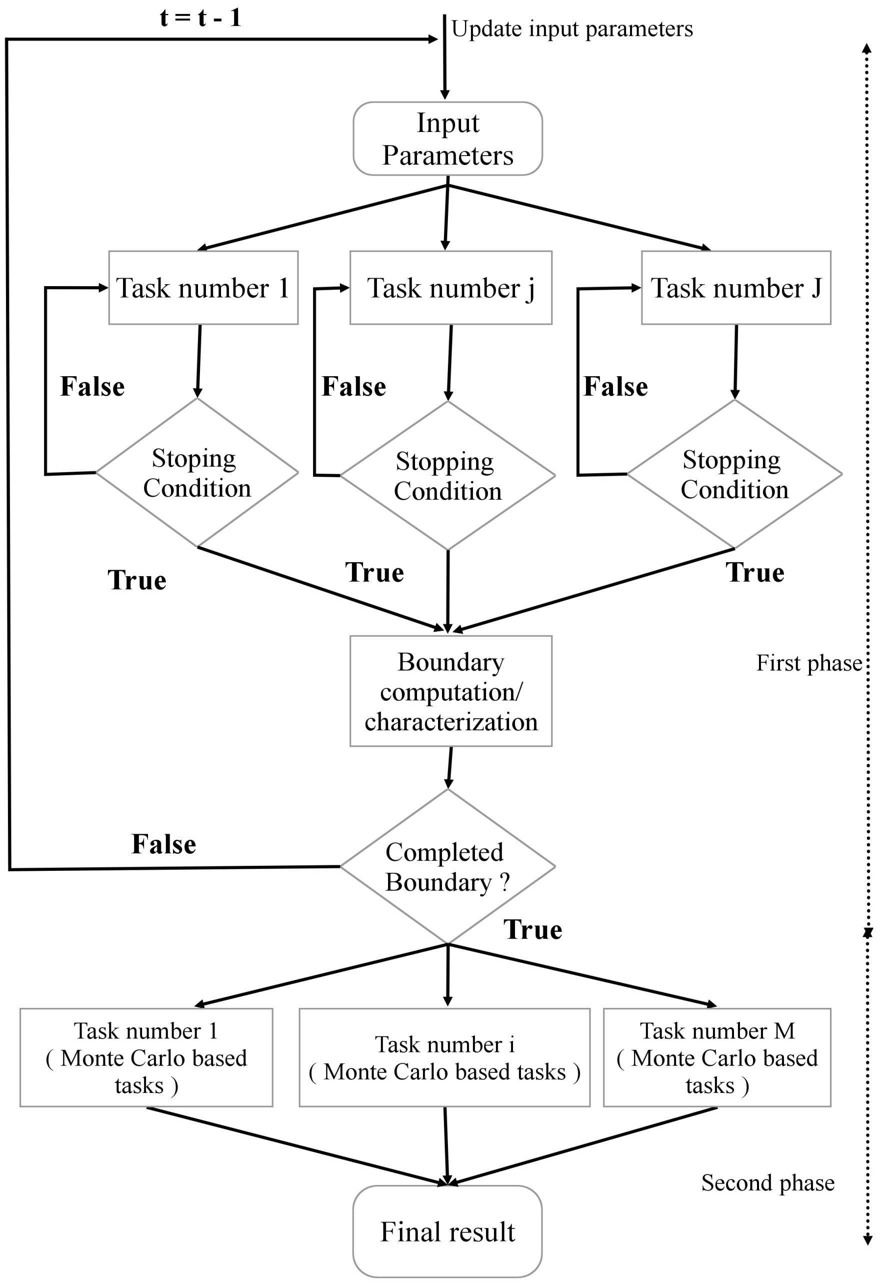

Figure 2.3: Generic Flowchart of Monte Carlo Based American Option Pricing Algorithms

that an object deployed and running on the grid may, during its lifetime, switch from playing the role of a worker or of a master and vice-versa.

The simulators are managed by a leader, which is called *controller* and has the responsibility to control the execution of the pricing application: at various stages it sets up simulators which include one or several sets of a master plus its workers, to complete the various steps of the computation.

A master does not orchestrate the computation, it is just a "service" which responds to requests from the controller and from its workers. The controller does not expect the results for allocated work through a classical request-reply blocking mechanism. Instead, the controller polls regularly a flag, under the master(s) responsibility, in order to be informed when the current Master/Worker round is terminated. Then, the controller knows it can read the result on master(s), and use it to trigger the next step of the algorithm. For the next step, the controller can decide how to proceed, either reusing the simulators which have been created (with some new parameters/setup), or having them drop out of scope (and be garbage collected). So, in the next step, a simulator is not mandatorily playing the same role as before, and, in case it is a worker, it may rely on a different master object previously.

Workers act as independent and asynchronous "agents" who request additional work from their master as soon as their work queue becomes empty. They go idle when their master has no work to give them.

The figure below illustrates this approach on a case with only one master.

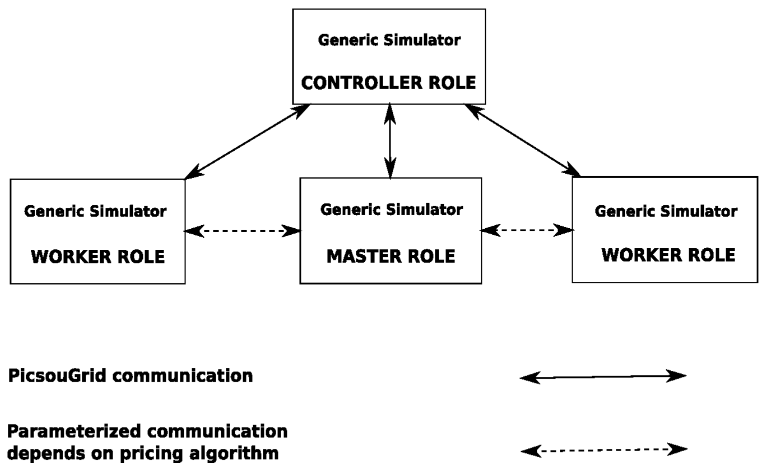

Figure 2.4: Combined Master/Worker Hierarchy - Case with only one Master-Workers set

2.4.2.2 Flexibility

Let us come back to the algorithm presented in Figure 2.3 to illustrate that the new architecture is able to support Master/Worker applications requiring different parallel phases, phases requiring themselves different task definition. The tasks in the first phase aim to compute/characterize the exercise boundary. These tasks are treated independently, each one has one index number j with $j = 1, \dots, J$. In fact, each task needs to finish some iterative cycles to converge, thus the computational time for each task is not unique. For that reason the master always gets the soonest arrival task results, merges them to the total result. Then the master increases the task index number, distribute this new task to an idle worker, until it finishes all J tasks. The advantage of this approach is that the fastest machine is always the most mastered worker. In case of failure, the master may not care about this crash. However, once the master figures out a missing task by verifying the completed task index, it will automatically re-generate it then re-distribute it to the idle worker requiring some work.

The second phase of this kind of algorithm aims to compute the option prices and it consists of a batch of tasks of Monte Carlo simulations. Therefore the distributed mechanism in this phase works similarly as a distributed European option pricing. It can be computed in parallel, typically in blocks of 10^3 to 10^4 simulations, and then the statistics gathered to estimate option prices, but without needing to identify tasks by an index.

2.5 Programming Supports For PicsouGrid

2.5.1 ProActive Programming Model

The PicsouGrid framework has been developed in Java with the ProActive [11] parallel and distributed computing library. The use of Java allows our framework to be used in a wide range of computing environments, from standard Windows desktop systems, to large Linux clusters standalone, or part of a multi-cluster grid. ProActive implements the *Active Object* model to enable concurrent, asynchronous object access and guarantees of deterministic behaviour. Incorporating ProActive into PicsouGrid has minor impact on the framework or specific algorithm implementations. ProActive imposes a few constraints on the construction of Objects which will be accessed concurrently, such as empty argument constructors, limited use of self reference in method call parameters, no checked exceptions, and non-primitive, non-final return values for methods. In return, ProActive provides a generic object factory which will dynamically instantiate a "reified" version of any desired object on any available host, while providing the application with a stub which can be utilised exactly as an instance of the standard object. The reified object consists of the proxy stub, a wrapper object, a message queue, and a wrapped instance of the actual object. Only the proxy stub is on the local node. The wrapper object is started by ProActive on the remote node (either specified explicitly as a parameter to the object factory, or selected automatically by ProActive), and contains a message queue for all public method calls on the object, and finally the wrapped object itself. PicsouGrid makes heavy use of the Typed Group Communications features of ProActive[10] to trigger the parallel processing of Monte Carlo simulations on workers, as well as broadcast and gather of associated parameters for uniform configuration and interrogation of worker object states. Through the use of ProActive it is possible to run simulations on a single machine, a desktop grid, a traditional cluster, or on a full grid environment without any additional configuration effort in the application. The ProActive deployment mechanism automatically contacts and initiates services and objects on the remote nodes [11].

2.5.2 The ProActive/GCM Deployment model

grid environment raises a lot of challenges for deploying problem because it consists in running applications over large-scale heterogeneous resources that evolve dynamically.

The deployment of PicsouGrid applications throughout ProActive runtime deployment is based on the ProActive/GCM model deployment. The main idea behind this model is to fully eliminate from the source code the following elements: machine names, creation protocols and registry/lookup protocols, so that users could deploy applications on different computing resources, without changes in the source code.

On the application context, the deployment model relies on the notion of *Virtual Nodes (VNs)* which can be composed by one or more nodes, each node representing a ProActive runtime deployed on a physical (or virtual in the case of virtual machines) resource.

The whole deployment process (Fig. 2.5) and environment configuration is defined by means of XML descriptors which depict the application requirements and deployment process. The deployment of ProActive/GCM applications depends on two types of descriptors:

2.5.2.1 GCM Application Descriptors (GCMA) :

The GCMA descriptors define applications-related properties, such as localization of libraries, file transfer, application parameters and non-functional services (logging, security and checkpoint). The resources requirement is also defined, but taking the VNs and nodes into account. Besides, the GCMA defines one or multiple resource providers.

2.5.2.2 GCM Deployment Descriptors (GCMD) :

The GCMD descriptors define the operation of the resource providers. This includes the access protocols to reach the resources (e.g. SSH, RSH, GSISSH, etc.), resource reservation protocols and tools which are sometimes required to have access to resources (e.g. PBS, LSF, Sun grid Engine, OAR, etc.), process (e.g. JVM) creation protocols which have a relation on how to launch processes (e.g. SSH, OAR, gLite, Globus) and communication protocols (e.g. RMI, HTTP, SOAP, etc).

The need of these two kinds of descriptors enforces a clear separation between application definition and deployment process. The advantages of this model are clear: on one side users want to add a new resource provider (e.g. a private cluster, production grid), the application code does not change and a single line is enough to add the resource provider to the application descriptor (GCMA). On the other side, the definition of the deployment process happens just once for each resource and can be reused for different applications.

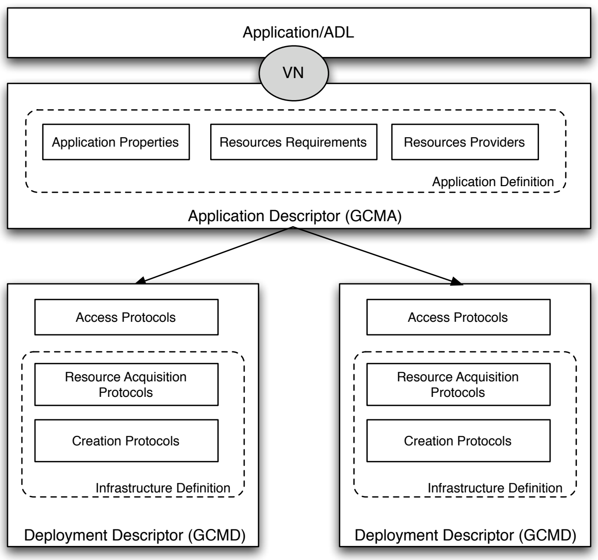
Application/ADL
VN
Application Properties
Resources Requirements
Resources Providers
Application Definition
Application Descriptor (GCMA)
Access Protocols
Resource Acquisition Protocols
Creation Protocols
Infrastructure Definition
Deployment Descriptor (GCMD)
Access Protocols
Resource Acquisition Protocols
Creation Protocols
Infrastructure Definition
Deployment Descriptor (GCMD)

Figure 2.5: GCM Descriptor-based deployment model

2.5.3 SSJ: Stochastic Simulation in Java

SSJ is a Java library for stochastic simulation, developed under the direction of Pierre L'Ecuyer, in the "Departement d'Informatique et de Recherche Operationnelle" (DIRO)

[74], at the Université de Montreal. It provides facilities for generating uniform and non-uniform random variables, computing different measures related to probability distributions, performing goodness-of-fit tests, applying quasi-Monte Carlo methods, collecting (elementary) statistics, and programming discrete-event simulations with both events and processes.

SSJ package provides *RandomStream* as an interface that defines the basic structures to handle multiple streams of uniform random numbers and convenient tools to move around within and across these streams. In SSJ package, the actual PRNGs are provided in classes that implement the *RandomStream* interface. Each stream of random number is an object of the class that implements such interface and can be viewed as a virtual random number generator. For each type of base PRNG, the full period of the generator is cut into adjecent streams (or segments) of length Z and each of these streams is partitioned into V sub-streams of length W, where hence $Z = VW$. The values of V and W depend on the specific PRNG, but are usually larger than 2^{50}. Thus, the distance Z between the starting points of two successive streams provided by an PRNG usually exceeds 2^{100}. The initial seed of the PRNG is the starting point of the first stream. It has a default value for each type of PRNG, but this initial value can be changed by calling *setPackageSeed* method for the corresponding class. Each time a new *RandomStream* is created, its starting point (initial seed) is computed automatically, Z steps ahead of the starting point of the previously created stream of the same type, and its current state is set equal to this starting point.

As an example, we consider a generator, that we decided to use in PicsouGrid, the *MRG32k3p* generator. The *MRG32k3p* generator is a combined multiple recursive one, proposed by L'Ecuyer and Touzin (2000) [75], implemented in 32-bit integer arithmetic. The generator has a period length of $\rho \approx 2^{185}$. The value V, W and Z are $2^{62}, 2^{72}$ and 2^{134}, respectively. For each stream, one can advance by one step and generate one value, or go ahead to the beginning of the next sub-stream within this stream, or go back to the beginning of the current sub-stream, or to the beginning of the stream, or jump ahead or back by an arbitrary number of steps.

2.6 PicsouGrid Implementation and Usage

2.6.1 PicsouGrid Framework Core Description

Figure 2.6 presents the package diagram of PicsouGrid framework. The core of PicsouGrid is stored in the package *core*. The package *data* includes the description of data and data type description such as underlying asset price, algorithm parameters etc. The packages *util* and *gui* contain the useful tools, graphic user interfaces respectively. Pricing algorithm classes are stored in *pricing* package.

In terms of the object hierarchy, there are two parts: one for the grid aspects *Pic-*

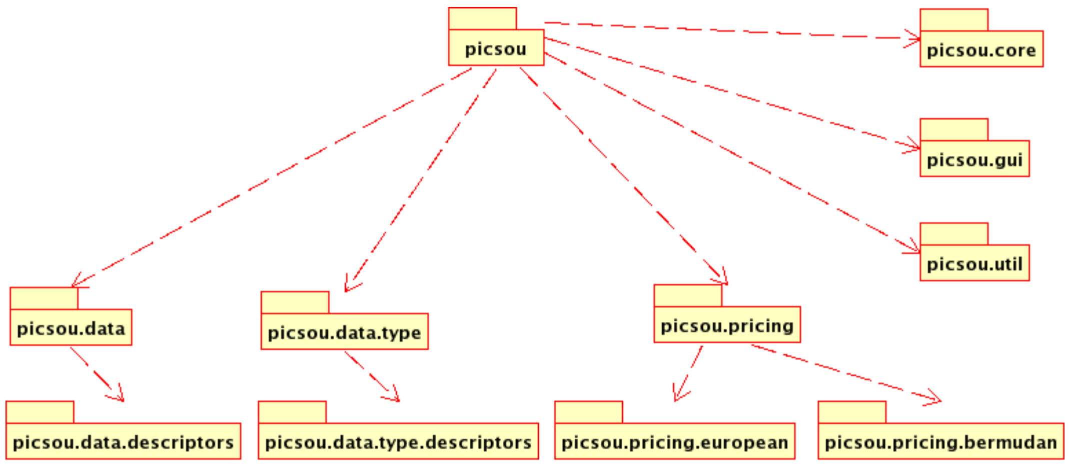

Figure 2.6: PicsouGrid Package Diagram

souGridInterface and one which captures the algorithmic aspects *SimulatorInterface*, then an abstract class which implements several common methods called *SimulatorAbs*. Concrete algorithm simulators then inherit from this, either directly or indirectly (in the case where more intermediate abstract classes are required for similar classes of algorithms). We provide an idea of the core object relationships in Figure 2.7.

In order to allow PicsouGrid handling multiple even if successive option pricings per execution, we define 4 states in PicsouGrid as follow: *INIT, READY, RUNNING, DONE*. The *INIT* state is established whenever PicsouGrid application has been launched. During this state, the system has the responsibility to create the master, workers and to deploy them on computing environment. Once workers have been successfully deployed, the system will pass to the *READY* state. At this state, workers are ready to receive any job. The state will be changed to *RUNNING* whenever the workers receive their first tasks and is unchangeable during the computation. After the master collected enough simulations to compute the option price, the state will be set to *DONE*. The system will complete the rest of computation by saving the estimated results in an output file or on screen. Then the state will be now reset to *READY* in case there exists another coming computing requirement otherwise it remains in the *DONE* state.

- The *PicsouGridInterface* interface provides the helper methods for handling grid aspects in PicsouGrid framework. The first one is *setWorkers* method that has the responsibility to add a set of workers to a given *Simulator*. This method can only be done while *Simulator* is in *INIT* state, as all subsequent operations will be cascaded to workers, if they exist. The entry and exit state are both *INIT*.

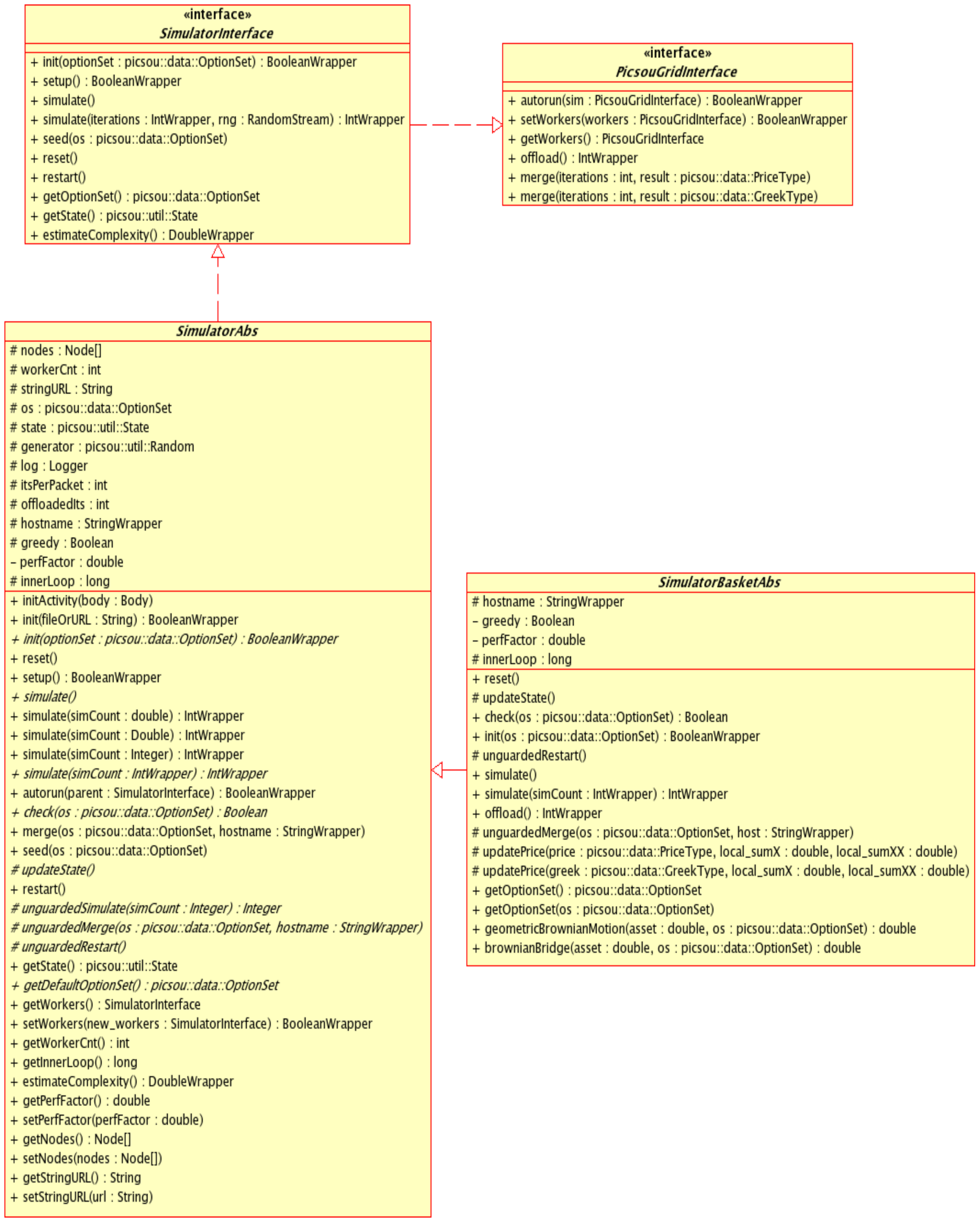
«interface»
SimulatorInterface
+ init(optionSet : picsou::data::OptionSet) : BooleanWrapper
+ setup() : BooleanWrapper
+ simulate()
+ simulate(iterations : IntWrapper, rng : RandomStream) : IntWrapper
+ seed(os : picsou::data::OptionSet)
+ reset()
+ restart()
+ getOptionSet() : picsou::data::OptionSet
+ getState() : picsou::util::State
+ estimateComplexity() : DoubleWrapper
«interface»
PicsouGridInterface
+ autorun(sim : PicsouGridInterface) : BooleanWrapper
+ setWorkers(workers : PicsouGridInterface) : BooleanWrapper
+ getWorkers() : PicsouGridInterface
+ offload() : IntWrapper
+ merge(iterations : int, result : picsou::data::PriceType)
+ merge(iterations : int, result : picsou::data::GreekType)
SimulatorAbs
# nodes : Node[]
# workerCnt : int
# stringURL : String
# os : picsou::data::OptionSet
# state : picsou::util::State
# generator : picsou::util::Random
# log : Logger
# itsPerPacket : int
# offloadedIts : int
# hostname : StringWrapper
# greedy : Boolean
– perfFactor : double
# innerLoop : long
+ initActivity(body : Body)
+ init(fileOrURL : String) : BooleanWrapper
+ init(optionSet : picsou::data::OptionSet) : BooleanWrapper
+ reset()
+ setup() : BooleanWrapper
+ simulate()
+ simulate(simCount : double) : IntWrapper
+ simulate(simCount : Double) : IntWrapper
+ simulate(simCount : Integer) : IntWrapper
+ simulate(simCount : IntWrapper) : IntWrapper
+ autorun(parent : SimulatorInterface) : BooleanWrapper
+ check(os : picsou::data::OptionSet) : Boolean
+ merge(os : picsou::data::OptionSet, hostname : StringWrapper)
+ seed(os : picsou::data::OptionSet)
# updateState()
+ restart()
# unguardedSimulate(simCount : Integer) : Integer
# unguardedMerge(os : picsou::data::OptionSet, hostname : StringWrapper)
# unguardedRestart()
+ getState() : picsou::util::State
+ getDefaultOptionSet() : picsou::data::OptionSet
+ getWorkers() : SimulatorInterface
+ setWorkers(new_workers : SimulatorInterface) : BooleanWrapper
+ getWorkerCnt() : int
+ getInnerLoop() : long
+ estimateComplexity() : DoubleWrapper
+ getPerfFactor() : double
+ setPerfFactor(perfFactor : double)
+ getNodes() : Node[]
+ setNodes(nodes : Node[])
+ getStringURL() : String
+ setStringURL(url : String)
SimulatorBasketAbs
# hostname : StringWrapper
– greedy : Boolean
– perfFactor : double
# innerLoop : long
+ reset()
# updateState()
+ check(os : picsou::data::OptionSet) : Boolean
+ init(os : picsou::data::OptionSet) : BooleanWrapper
# unguardedRestart()
+ simulate()
+ simulate(simCount : IntWrapper) : IntWrapper
+ offload() : IntWrapper
# unguardedMerge(os : picsou::data::OptionSet, host : StringWrapper)
# updatePrice(price : picsou::data::PriceType, local_sumX : double, local_sumXX : double)
# updatePrice(greek : picsou::data::GreekType, local_sumX : double, local_sumXX : double)
+ getOptionSet() : picsou::data::OptionSet
+ getOptionSet(os : picsou::data::OptionSet)
+ geometricBrownianMotion(asset : double, os : picsou::data::OptionSet) : double
+ brownianBridge(asset : double, os : picsou::data::OptionSet) : double

Figure 2.7: PicsouGrid Core Classes Diagram

```
public BooleanWrapper setWorkers(PicsouGridInterface workers);
```

The second method is *getWorkers* that returns the set of *Simulator* sub-workers associated with this *Simulator*. It can be called at any time (i.e. with any type of states) and may return "null".

```
public PicsouGridInterface getWorkers();
```

The *autorun* method asks the simulator to run, pulling tasks from simulators and merging them back as they complete. This method returns "true" when done. This is a ProActive artifact, since the returned object allows ProActive to hold a future to synchronize on [11]. The entry and exit state are both not *INIT*.

```
public BooleanWrapper autorun(PicsouGridInterface sim);
```

The last method *merge* has the responsibility to add the results of another simulator to the *Simulator* which acts as a master.

```
// merging results for option pricing
public void merge(IntWrapper simCount, PriceType result);
// merging results for Greek hedging
public void merge(IntWrapper simCount, GreekType result);
```

The pre-condition on simulators claims that the states are either *RUNNING* or *DONE*. Meanwhile the pre-conditions on the master *Simulator* claims that the state is either *READY* or *RUNNING* and satisfying the following expression:

```
Simulator master = new Simulator();
master.completedSimulations + completedSimulations <= master.totalSimulations
```

The entry state is either *READY* or *RUNNING* while the exit state is either *RUNNING* or *DONE*.

- The *SimulatorInterface* interface provides the helper methods to capture the algorithmic aspects. The *init* method helps to set up the *Simulator* with a pricing algorithm and basket of assets. The entry state is *ANY* while the exit state is *READY*.

  ```
  public BooleanWrapper init(OptionSet optionSet);
  ```

  The *setup* method does initial setup calculations. Both entry and exit states are *READY*.

```
public BooleanWrapper setup();
```

The *simulate* method does a simulation on the basket of assets with the given parameters and using all remaining *iterationsPerEstimationValuePacket*. The entry state is either *READY* or *RUNNING* and the exit state is *DONE*.

```
public void simulate();
```

The *simulate* method with number of iterations and a stream of random numbers attempts to simulate up to "*iterationsPerEstimationValuePacket*", return actual number completed. Notice that the simulator will not simulate more than the total *iterationsPerEstimationValuePacket* set by the algorithm. The entry state is either *READY* or *RUNNING* and the exit state is either *RUNNING* or *DONE*.

```
public IntWrapper simulate(IntWrapper iterations, RandomStream rng);
```

The *reset* and *restart* methods will reset the *Simulator* to initial state and clear the current results to restart the *Simulator* with the original algorithm and basket of assets (including clearing both of performance and numerical results) respectively. Thus, the *reset* method gets the entry state as ANY, returns the exit state as INIT and the *restart* method gets the entry state as not *INIT*, returns the exit state as *READY*.

```
public void reset();
public void restart();
```

The *estimateComplexity* method uses the configuration details to estimate the complexity of the pricing request with a specific algorithm. The entry state is not *INIT* and the exit state is unchanged.

```
public DoubleWrapper estimateComplexity();
```

2.6.2 Developer View Point

In this section, we are going to detail the steps that someone willing to implement a new pricing algorithm has to follow. The combined Master/Worker hierarchy relies strongly on the group communication mechanism that is built upon the ProActive elementary mechanism for asynchronous remote method invocation with automatic future for collecting a reply. As this last mechanism is implemented using standard Java, the group mechanism is itself platform independent and must be thought of as a replication of more than one (say

N) ProActive remote method invocations towards N active objects. Of course, this mechanism incorporates some optimizations, in such a way as to achieve better performances than a sequential achievement of N individual ProActive remote method calls. The availability of such a group communication mechanism in the ProActive library that we are relying upon, simplifies the programming of applications with similar activities running in parallel. Indeed, from the programming point of view, using a group of active objects of the same type, subsequently called a typed group, takes exactly the same form as using only one active object of this type. This is possible due to the fact that the ProActive library is built upon reification techniques: the class of an object that we want to make active, and thus remotely accessible, is reified at the meta level, at runtime. In a transparent way, method calls towards such an active object are executed through a stub which is type compatible with the original object. The stub's role is to enable to consider and manage the call as a first class entity and applies to it the required semantics: if it is a call towards one single remote active object, then the standard asynchronous remote method invocation of ProActive is applied; if the call is towards a group of objects, then the semantics of group communications is applied.

In this section we will illustrate how the use of ProActive active object and group communication is relevant for developing parallel pricing algorithm in PicsouGrid framework. We will take an example of a high dimensional American option pricing algorithm originally provided by Ibanez and Zapatero (2004) [63] and parallelized by Muni Toke (2006) [92]. Notice that all the classes that we present below are stored in *picsou.pricing* package. To handle in a sinple way the usage of grid nodes, we can create a class (e.g. *BenchWith-ProActive*, see Figure 2.8) mainly in charge of initiating the grid resources acquisition and deployment of computing entities on them.

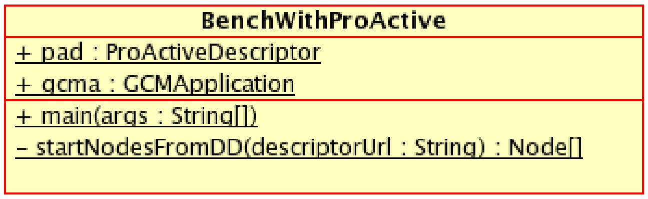
BenchWithProActive
+ pad : ProActiveDescriptor
+ gcma : GCMApplication
+ main(args : String[])
– startNodesFromDD(descriptorUrl : String) : Node[]

Figure 2.8: BenchWithProActive Class

Such class has the responsibility to receive the GCM Deployment Descriptor in order to acquire grid computing resources remotely accessible through the notion of ProActive nodes.

```
private static Node[] startNodesFromDD(String descriptorUrl) {
        Node[] nodes = null;
        try {
            File descriptor = new File(descriptorUrl);
            pad = PADeployment.getProactiveDescriptor(descriptorUrl);
            pad.activateMappings();
            nodes = pad.getVirtualNodes()[0].getNodes();
        } catch (NodeException ex) {
            ex.printStackTrace();
        } catch (ProActiveException ex) {
            ex.printStackTrace();
        }
        return nodes;
}
```

Regarding the pricing algorithm aspects, we consider the following steps

1. Controller Creation: This is the very first mission. According to the architecture presented in Figure 2.4, the controller has the responsibility to control the execution of option pricing applications. As controller acts as a leader simulator, but still is a simulator because it has to participate in the orchestration of the parallel pricing process, it inherits common methods from the abstract *SimulatorBasketAbs*.

   ```
   // for an American option pricing application
   SimulatorBasketAbs controller = new American();
   Node[] paNodes = startNodesFromDD(args[0]);
   controller.setNodes(paNodes);
   controller.init(optionSet);
   controller.simulate();
   ```

2. Master/Worker Creation : Under the controller supervision, the programmer has to define master and worker methods. Regarding the algorithm provided by Ibanez and Zapatero, we name the class gathering such methods *IZBskWorker*. This *IZBskWorker* class includes both master's methods such as merging method and the partial calculation methods of workers. The master is created as a ProActive active object,

   ```
   // At the controller side we do
   master = (IZBskWorker)PAActiveObject.newActive(
           IZBskWorker.class.getName(),new Object[] {});
   ```

   A group of workers will be created by using a static method of ProActive active objects group mechanism

```
workers = (IZBskWorker)
                PAGroup.newGroup(IZBskWorker.class.getName(),
                new Object[] {}, getNodes());
```

Workers can be created empty and existing active objects can be added later as described below. Non-empty workers can be built at once using two additional parameters: a list of parameters required by the constructors of the object members of the group, and a list of nodes where to map those members. In that case the group is created and new active objects are constructed using the list parameters and are immediately included in the group. The n^{th} active object is created with the n^{th} parameter on the n^{th} node. If the list of parameters is longer than the list of nodes (i.e. we want to create more active objects than the number of available nodes), active objects are created and mapped in a round-robin fashion on the available nodes. We can access the individual members of the group by using the following piece of code, e.g. :

```
PiBskWorker worker0 = (IZBskWorker) workers.get(0);
PiBskWorker worker1 = (IZBskWorker) workers.get(1);
```

3. Tasks Definition and Submission: The specification of tasks to be further allocated to workers depends on the type of option pricing. According the Ibanez and Zapatero algorithm, the first phase's duty is building the exercise boundary function for each asset using regression at every opportunity dates. At each opportunity date, we have to compute a finite number of boundary points (J points) called optimal boundary points to regress the boundary function. Each point computation can be done separately and independently. Thus normally we have J independent tasks to do in parallel. Controller will allocate implicitly one task per worker. Once a worker finishes its computation and sends results back to master, then the master will automatically assign the next task to this idle worker. For the tasks distribution, we can use a polling mechanism: the controller must sleep and wait for the master to collect and merge results, polling (e.g. 5 times per second (200ms sleep)) to check if the stopping condition is true. Once gathering enough J points, the controller performs a regression to obtain the boundary function and add it to the boundary set.

   Based on a backward dates computing, earlier estimated functions will be used for later dates, so for each opportunity date the controller needs to fulfil the workers with current updated functions *boundarySet*. The computation methods of the first phase task are implemented within the *computeFirstPhaseTasks* method. Then tasks are submitted to workers through classes by calling this execution method. Regarding the execution method towards a group of objects, the default behaviour is to broadcast these tasks to all workers. The following piece of code at the controller side corresponds to this first phase process (in backward computing)

```
// START FIRST PHASE
// At the controller side
// Backward steps computing
for (t = TOTAL_NUMBER_OPPORTUNITIES - 1; t > 0; t--) {
        for(d = 1; d < TOTAL_NUMBER_ASSETS; d++){
                workers.setupWorkers(t, boundarySet);
                // Submit implicitly one task per worker.
                // Notice to each worker its reference master
                // for further merging results.
                workers.computeFirstPhaseTasks(getMaster());
                while master.getCompletedPoints() < J {
                try {
                        Thread.sleep(200);
                        } catch (Exception ex) {
                        ex.printStackTrace();
                        }
                }
        // Perform a new boundary function model and add it to the boundary set
        boundarySet.add(boundaryRegression(master.getJPoints()));
        }
}
// END FIRST PHASE
```

We detail the *computeFirstPhaseTasks()* method for the first phase: After finishing a computation of boundary point, the master (which is represented by the reference *getMaster()*) will be asked in order for it to accumulate such computed point, see below the piece of code running in parallel on each worker:

```
// At the parallel workers side
// The parent represents the master reference
public BooleanWrapper computeFirstPhaseTasks(PicsouGridInterface parent) {
        // get the assigned task index (or point index)
        int idx = parent.getNextBoundaryPoint().intValue();
        while (idx >= 0){
                calcBoundaryPoints();
                // Add the computed point to point set.
                parent.merge();
                // the merge() method implicitly increases
                // the number of computed point by 1.
                idx = parent.getNextBoundaryPoint().intValue();
        }
return new BooleanWrapper(true);
}
```

And here is the detail of *getNextBoundaryPoint()* method:

```
// Set the next point index
public IntWrapper getNextBoundaryPoint() {
        int idx;
        if (nextBoundaryPointIdx < J) {
                idx = nextBoundaryPointIdx;
                nextBoundaryPointIdx += 1;
        } else {
                idx = -1;
        }
        return new IntWrapper(idx);
}
```

and *calcBoundaryPoints()* method (each point require a number of iterations for converge):

```
private BooleanWrapper calcBoundaryPoints() {
        while (stop == false) {
                // estimate the optimal boundary point
                simulate();
                // if converge reached then
                Stop = true;
        }
return new BooleanWrapper(true);
}
```

Once the algorithm finishes the first phase, then it starts the second phase. The second phase consists of straightforward Monte Carlo simulations (e.g. $nbMC$ is the total number of simulations), starting at opportunity zero. It is enough for the controller to fulfil the workers with complete boundary set just before the beginning of the second phase. In parallel, each worker will execute a partial number of Monte Carlo simulations (e.g. $nbMC/P$ simulations, where P is the number of workers). Master has the responsibility to merge such simulated results. Finally, controller will compute the final results.

```
// START SECOND PHASE
// At the controller side
// Starting at t = 0
// fulfil the workers with ‘‘boundarySet’’ variable
workers.setupWorkers(0, boundarySet);
workers.computeSecondPhaseTasks(getMaster());
while (master.getCompletedIterations() < TOTAL_NUMBER_ITERATIONS_SECOND_PHASE)
{
      try {
           Thread.sleep(200);
          } catch (Exception ex) {
           ex.printStackTrace();
          }
}
// get the final results
totalSumResult = master.getSumResult();
totalSumSquareResult = master.getSumSquareResult();
// compute and display the prices (i.e. a call option)
updatePrice(os.getBasket().getCallPrice(), totalSumResult, totalSumSquareResult);
// END SECOND PHASE
```

For further details, this piece of code shows the *computeSecondPhaseTasks()* method corresponding to the second phase:

```
// At the parallel workers side:
// The parent represents the master reference
public BooleanWrapper computeSecondPhaseTasks(PicsouGridInterface parent) {
        for (t = 1; t < TOTAL_NUMBER_OPPORTUNITIES; t++) {
                for(d = 1; d < TOTAL_NUMBER_ASSETS; d++){
                        simulate(iterations, rng);
                }
        }
        // merging partial simulated results and implicitly and
        // increasing the number of completed simulations by ‘‘iterations’’.
        parent.merge(iterations);
return new BooleanWrapper(true);
}
```

4. Resources Releasing: To terminate PicsouGrid, we will first destroy all active objects

then the JVMs in order to complete the cleanup. For destroying active objects, the master will first terminate all the workers and then itself by calling the following method.

```
PAActiveObject.terminateActiveObject()
```

Once active objects are destroyed, then JVMs must be killed. Alternatively, one can focus on killing the JVMs, that will get rid of the active objects too. ProActive gives the ability to kill all JVMs and Nodes deployed with an XML descriptor with the method: killall(boolean softly) in class ProActiveDescriptor.

```
ProActiveDescriptor
        pad = PADeployment.getProactiveDescriptor(String xmlFileLocation);
//Kills every jvm deployed through the descriptor
pad.killall(false);
```

2.7 PicsouGrid Experiments on the Grid

This section has two objectives. The first objective consists in understanding the latencies and behaviour within the various layers of the grid environment on which PicsouGrid applications were being run. Meanwhile the second one aims to evaluate the performance, fault tolerance and tasks distribution of PicsouGrid framework, experimented using basket European option pricing on a grid environment.

In case of the first goal, it became clear that a common model was necessary in order to understand the latencies and behaviour of the grid environment. Hence we first define terms related to our grid process model and present a recursive state machine which has been used for tracking the life cycle of grid jobs. Then we only focus on establishing a testcase of parallel simulations on the grid (e.g. parallel European option pricing using Monte Carlo method), and as such do not focus on parallel speed-up *per se*. To simplify the experiments and to highlight the issues introduced by a grid infrastructure and parallel computing environment used to access grid machines, we have only executed the Monte Carlo stage of the parallel computation of the vanilla option pricing, and removed the synchronization at the end of the Monte Carlo stage. In this way, the experimental jobs appear to be embarrassingly parallel.

In case of the second goal, we will present the good performance results we obtained for a basket European option pricing implemented using PicsouGrid framework, running on a single cluster and then on a multi-site grid. The performance results show good speedup. Other features of PicsouGrid such as tasks distribution and fault tolerance will be also evaluated.

2.7.1 Grid Process Model

The Unix process model [109], with its three primary state of READY, RUNNING, and BLOCKED, provides a common basis for the implementation of POSIX (Portable Operating System Interface for UniX) operating system kernels, and an understanding of the behaviour of a process in a pre-emptive multi-tasking operating system. Users and developers have a clear understanding of the meaning of system time (time spent on kernel method calls), user time (time spent executing user code), and wait/block time (time spent blocking for other processes or system input/output). There are analogous requirements in a grid environment where users, developers, and system administrators need to understand what state and stage a "grid job" is in at any given time, and from a perspective be able to analyze the full job life-cycle. The federated nature and automated late-allocation of disperse and heterogeneous computing resources to the execution of a grid job make it difficult to achieve this.

2.7.1.1 Grid Tasks and Jobs

There is no commonly agreed model for a task in a grid environment. As a result, it is difficult to discuss and design systems which manage the life-cycle of a program executing on a grid. This is partially due to the lack of a common definition of a "grid task", and its scope. The GGF grid Scheduling Dictionary [98] offers two short definitions which provide a starting point:

- **Job**: An application or task performed on High Performance Computing resources. A Job may be composed of steps/sections as individual schedulable entities.

- **Task**: A specific piece of work required to be done as part of a job or application.

Besides the ambiguity introduced by suggesting that a *job* is also a *task*, these definitions do not provide sufficient semantic clarity for distinguishing between workflows and parallel executions. We therefore feel it is necessary to augment these terms to contain the concept of co-scheduling of resources to provide coordinated parallel access, possibly across geographically disperse resources. We propose the following definitions:

- **Basic Task** : A specific piece of work required to be done as part of a job, providing the most basic computational unit of the job, and designed for serial execution on a single hardware processor. The scheduling and management of a *basic task* may be handled by the grid infrastructure, or delegated to a *grid job*.

- **Grid Job** : A task or a set of tasks to be completed by a grid infrastructure, containing meta-data to facilitate the management of the task both by the user and a specific grid infrastructure.

- **Workflow Job** : A *grid job* containing a set of dependencies on its constituent basic tasks or other *grid jobs* and which is responsible for the coordinated execution and input/output management of those sub-jobs or sub-tasks.

- **Parallel Job** : A *grid job* which requires the coordinated parallel execution and communication of a set of *basic tasks.*

With this set of definitions we consider a simple parallel Monte Carlo simulation to consist of a *parallel job* with a set of coordinated *basic tasks*, such that the grid infrastructure provides a set of concurrent computing resources on which the simulation can initiate the parallel computation. A more complex phased Monte Carlo simulation would consist of a *workflow job* where each phase of the workflow would consist of a *parallel job*, executing that phase's parallel computation as part of the larger Monte Carlo simulation. The grid infrastructure is then responsible for the appropriate selection, reservation, and allocation/binding of the grid computing resources to the simulation job (whether simple or complex), based on the requirements described within the job itself.

2.7.1.2 Recursive Layered State Machine

Figure 2.9 indicates the system layers typically found in a grid environment and through which a grid job will execute. For a basic grid job, this will consist of one sub-process at each layer. It is possible that the *Site* layer will not always be present, with *Clusters* being accessed directly by the grid infrastructure. The visibility of a particular *Core*, in contrast to the *Host* in which it exists, also may not be distinguishable. Some clusters may allocate resources on a "per-host" basis, with all cores available for the executing task, while others may allocate a number of tasks to a particular host up to the number of physical cores available, trusting the host operating system to correctly schedule each grid task (executing as an independent operating-system-level process) to a different core. Finally, the concept of a *VM* (virtual machine), whether a user-level VM such as Java or an operating system level VM such as Xen or VMWare, either may not exist within the grid environment, or may replace the concept of a core, with one VM allocated to each core within the host, and the host (or cluster) then scheduling grid tasks on a "one-per-VM" basis.

It should be noted that Figure 2.9 only illustrates system level layers, predominantly representing the layers of physical hardware and networking. There are also the various

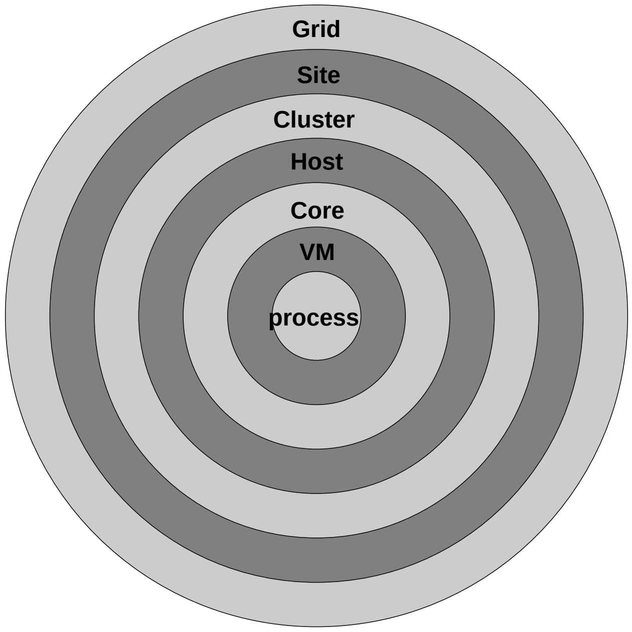

Figure 2.9: Various layers through which a grid job executes

layers of software, such as the grid framework, the local cluster management system, the operating system, and any application framework which may simultaneously be in use. In total, eight levels of nested software scripts and six levels of system infrastructure have been traversed in order to get from a single grid submit node to the thousands of distributed grid cores where the parallel compute job is finally executed.

When working in a cluster or grid environment, many of the aspects which can be easily assumed in a single node environment must be made explicit, and the staging of execution is managed with possible time delays for synchronization and queuing, and on completion it is necessary to properly return the collective results. Taking these various factors into consideration, a five-stage model is proposed which is applied at each layer of the grid infrastructure. The stages, in order, are defined as follows:

- **CREATE** prepares a definition of the process to be executed at this layer.

- **BIND** associates the definition, possibly based on constraints within the definition, with a particular system at this layer.

- **PREPARE** stages and data required for execution to the local system which the definition has been bound to and does any pre-execution configuration of the system.

- **EXECUTE** runs the program found in the definition. This may require recursing to lower layers (and subsequently starting from that layer's CREATE stage). In a parallel context, it is at this stage where the transition from serial to parallel execution takes place, possibly multiple times for the completion of this stage's execution.

- **CLEAR** cleans the system and post-processes any results for their return to the caller (*e.g.* next higher layer).

The grid infrastructure handles the transition from one stage to the next. To accommodate the pipelined and possibly suspended life-cycle of a grid job it is not possible to consider each stage as being an atomic operation. Rather, it is more practical to add entry and exit states to each stage. In this manner, three states are possible for each stage: READY, which is the entry state; ACTIVE, which represents the stage being actively processed; and DONE, when the stage has been completed and transition to the next stage is possible. A grid job starts in the CREATE.READY state, which can be seen as a "blank" grid job. Once the system or user has completed their definition of the actions for that layer (done by entering CREATE.ACTIVE), the grid job finishes in the CREATE.DONE state. At some later point, the grid infrastructure is able to bind the job to a resource, and later still prepare the bound node(s) for execution. When the node(s) are ready the grid job can execute, and finally, once the execution is complete, the grid job can be cleared.

We have developed this model to be applied recursively at the various layers shown in Figure 2.9. The layering also includes the software systems, and is arbitrary to a particular grid environment. For example, a grid job could be in the state "CLUSTER/BIND.READY", indicating that a cluster-level job description has been prepared, and now the grid job (or this portion of it) is waiting for the cluster layer of the grid infrastructure to make a binding decision to submit the job to a particular queue. The queue, in turn, would have to allocate the job to a particular host, and so on. While this model has been developed with the intention to incorporate it into a larger grid workload management system, the current model is only used for logging, time stamping, a monitoring. In many cases (*e.g.* Grid'5000 and EGEE) we do not have access to the internals of the grid infrastructure and either do not have visibility of some of the state transitions or are only able to identify state transitions and time stamps during post-processing of grid job logs made available once the job is complete.

This model has allowed us to gather behaviour and performance details for a consistent comparison between three key aspects of any parallel grid application: the grid infrastructure impact; the parallelization framework; and the core application code. It is the basis for all the monitoring and timing information which is provided in the results presented

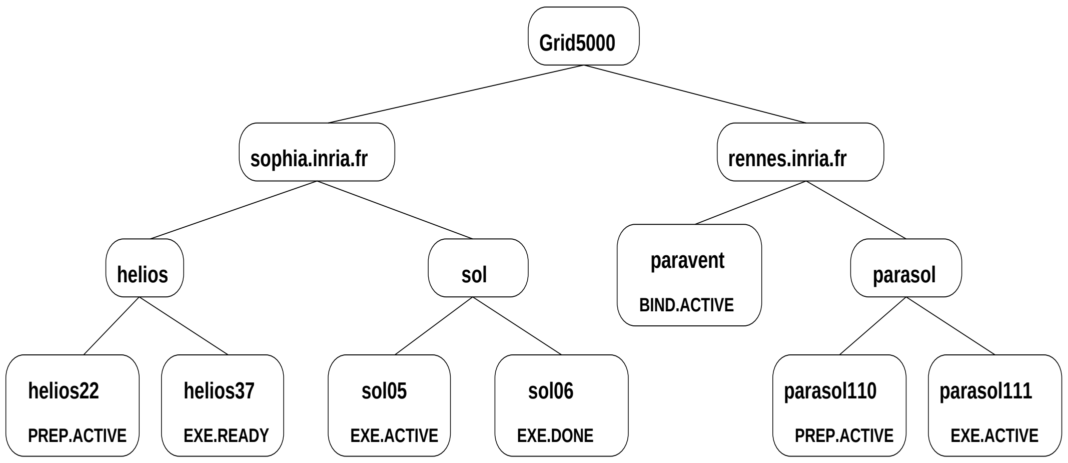

Figure 2.10: A simple example of a parallel grid job state snapshot on Grid'5000. Nodes without an explicit state are implicitly in the state EXE.ACTIVE, which is the only valid state for entering into a lower layer.

in Section 2.7.2. We finish this overview with Figure 2.10 which is a simplified grid network snapshot showing the state of various entities contributing to a fictitious small parallel computation. It shows two sites, each with two clusters. Three of the clusters have started executing the grid job on their worker nodes, and those six workers are in different states, while one cluster is making binding decisions regarding which workers to execute on.

2.7.1.3 Grid Efficiency Metrics Definition

We are going to define the metrics relevant to the experiment of a vanilla option pricing which appear to be embarrassingly parallel.

1. **Time window unit-job through put** : This metric counts the number of "unit jobs" executed by the grid infrastructure in a fixed time window. Typically the time window is taken from the start of the earliest computation to the end of the last computation, although this can be "time shifted" to align each cluster start-time as $t = 0$. If the number of grid nodes (processors/cores) is somehow fixed, this gives a comparative performance measure of the grid.

2. **Speed-up efficiency limit** : With some reference system serial calculation time for a unit job, the speed-up efficiency is defined as the time taken for the reference system to process N unit-jobs divided by the total occupancy time at a particular grid layer required to compute the same N unit-jobs. The metric assumes zero communication

time for parallel jobs. In this case the speed-up limit would always be N and the speed-up efficiency limit 1. Equation 2.1 defines this metric, where $n_{unitJob}$ is the total number of unit jobs completed by the grid system, n_{procs} is the number of processors contributing to the total computation time, and t_i represents the wall time of the occupancy of that layer of the grid.

$$\frac{n_{unitJob} \times t_{ref}}{\sum_{i=1}^{n_{procs}} t_i} \tag{2.1}$$

3. **Occupancy efficiency** : This measures what fraction of the total time available for computation was actually used by the application, measured at the various layers within the grid. This is defined by Equation 2.2, where $n_{compUnits}$ indicate the number of computational units (*e.g.* hosts, cores, VMs, threads, processes) available at that layer.

$$\frac{\sum_{i=1}^{n_{compUnits}} t_i}{n_{compUnits} \times t_{reservation}} \tag{2.2}$$

2.7.2 Experiment Analysis

2.7.2.1 Latency and Behaviour of the grid

The work presented here focuses on the capabilities and characteristics of the underlying grid infrastructure to provide for such application-level parallelism.

As mentioned earlier, to simplify the experiments and to highlight the issues introduced by the grid infrastructure and parallel computing environment, we only consider the parallel paths generation using Monte Carlo methods for vanilla option pricing, and removed the synchronization at the end. In this way, the experiments appear to be embarrassingly parallel. The starting point for discussing parallel Monte Carlo simulations on the grid is to understand an ideal situation. Ideally all available computing resources would be used at 100% of capacity for the duration of their reservation performing exclusively Monte Carlo simulations. The time to merge results would be zero, and there would be no communications overhead, latencies, or blocking due to synchronisation or bottlenecks.

In reality, as discussed in Section 2.7.1.3, there are many parameters which have an impact on the actual performance of a parallel Monte Carlo simulation. As the following results will show, predictable coordinated access to resources within a single cluster can be difficult, and synchronization of resources between clusters or sites even harder. Due to this observation, the work here focused on identifying the issues which lead to poor resource synchronization, and to facilitate evaluation of resource capability. In order to do this, the following results eliminate coordinated simulation and merging of results, and only initiate independent partial Monte Carlo simulations. All the following experiments were performed in early March 2007 on Grid'5000, using all available sites, clusters, and nodes. The figure headings show the statistics for the time spent in the states NODE.EXECUTE.ACTIVE and SIMULATOR.EXECUTE.ACTIVE in the form $node = (M, S)s$ and $sim = (M, S)s$ where M is the mean time in seconds and S is the standard deviation. $node_{eff}$ is the node occupancy efficiency, and sim_{eff} the simulator occupancy efficiency, as defined in Equation 2.2. The "simulator" is the part of the application where the Monte Carlo simulation is executed, excluding any software startup (e.g. JVM initiation), configuration etc.

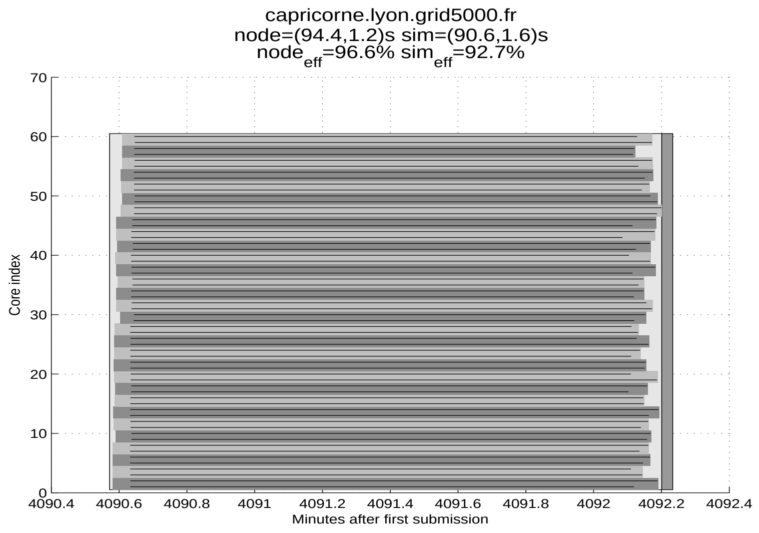

Figure 2.11: Realistic optimal parallel Monte Carlo simulation, with 92.7% occupancy efficiency.

Figure 2.11 shows an almost ideal situation where 60 cores running on 30 hosts from the same cluster all start processing within a second of each other, run their allocated simulations for the same duration (around 90 seconds, shown by the black "life line"), and complete in a time window of a few seconds. This provides a simulation efficiency of 92.7%, and we take this to be our "cluster-internal" efficiency standard.

By contrast, some clusters showed node (host) and core start and finish windows of several minutes, as seen in Figure 2.12. This particular example consists of 240 dual-CPU nodes, representing 480 cores. A simulation efficiency of only 43.1% was achieved, indicating that the resources were idle for the majority of the reservation time (the time outside of the black life-lines). Furthermore, the majority of this idle time was in the finishing window,

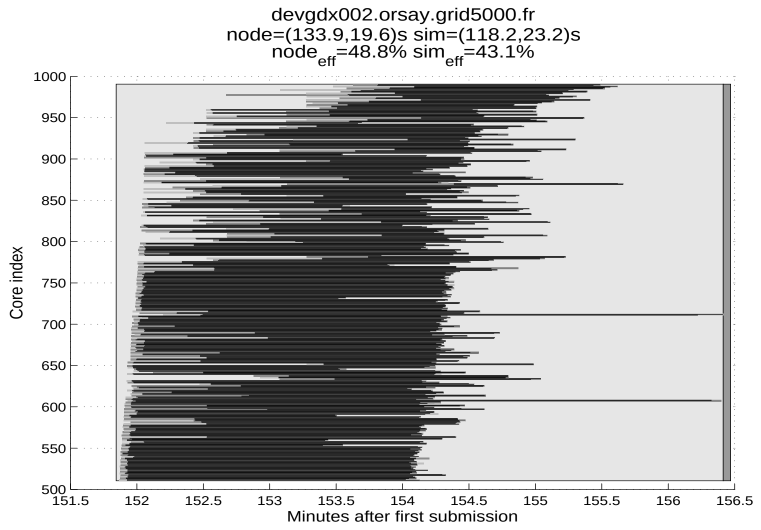

Figure 2.12: Cluster exhibiting significant variation in node start and finish times, resulting in an occupancy efficiency of $< 50\%$.

where only a few inexplicably slow nodes delayed completion of the computation on the node block within the cluster. The space prior to the start of the simulator life-line is due to grid infrastructure delays in launching the node- or core-level process (e.g. due to delays in the CLUSTER.PREPARE stage) – again, wasted computing time when the job held a reservation for the node and core, but failed to utilize it.

Now, we return to the question of coordinated multi-cluster (and multi-site) parallel computing. Besides inherent technical issues present when attempting regular communications across long network segments, it is difficult to satisfy on demand requests for grid-wide multi-node reservations. There is the initial challenge of immediate availability, and then the subsequent challenge of promptly completing and confirming the distributed node reservations and the requisite site, node, or cluster preparation (pre-execution configuration). Figure 2.13 shows an example of such a situation for a 1270-core multi-cluster job, where a request for approximately 80% of the reported available nodes from each cluster was made at a fixed point in time, and most clusters took over an hour before this request could be fulfilled. One cluster took two and a half days (not shown in figure due to effect on time scale). This is not a surprising result given the nature of Grid'5000 and the style of computations which are done on it, namely experimental (therefore usually less than one hour) parallel and distributed computing. At any given time it is expected that the clusters will be full with jobs which require blocks of nodes, and for the queue to contain multi-node jobs as well, therefore newly submitted jobs would expect to wait at least several hours to both make it to the front of the queue and for the requisite block of nodes to be available. While Grid'5000 provides simultaneous submission of jobs, it does not coordinate reservations and queues, so the individual clusters became available at different times. Using normalized cluster start-times, all the unit-job computations took place in a 274.2 second time window, for a total execute stage time block of 1270 cores $\times$ 274.2 seconds = 348234 core·seconds = 96.7 core·hours. Compared with a reference unit-job execution time of 67.3 seconds, the speed-up efficiency limit, as given in Equation 2.1, is (1270 cores $\times$ 67.3 seconds)/ 96.7 core·hours = 25.5%.

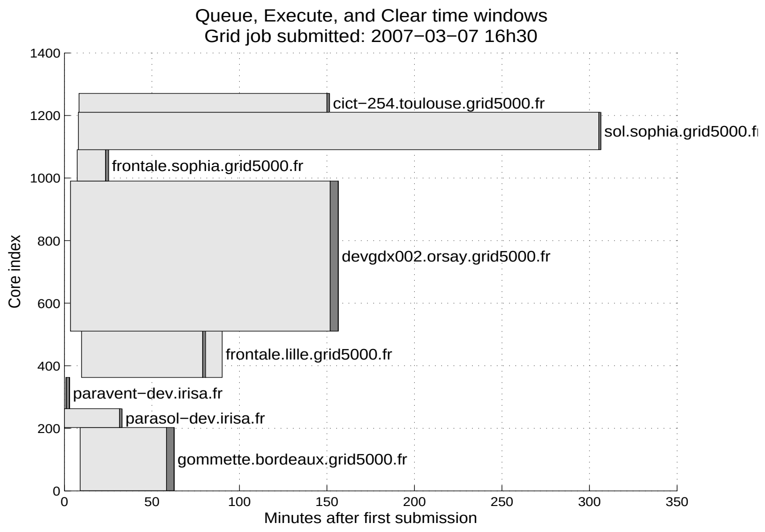

Figure 2.13: Light grey boxes indicate queuing or clearing time, dark grey boxes indicate execution time. This graph illustrates the difficulty in coordinated cross-site on-demand parallel computing.

Figure 2.14 shows the start-time aligned execution phase for each cluster. An obvious conclusion from this is the need to partition the computational load in such a way that the faster nodes are given more work, rather than remain idle while the grid job waits for the slowest nodes to complete.

A more realistic multi-cluster parallel grid job uses explicit cluster and node reservation. Figure 2.15 shows the results of a multi-cluster reservation for a large block of nodes per cluster at a fixed time of 6:05 AM CET (5:05 UTC), approximately 18 hours after the reservation was made. It was manually confirmed in advance that the requested resources should be available on all clusters, and a reduction in the number of nodes was made to provide for a margin of unexpectedly failed nodes. The 5 minute offset from the hour was to provide the grid and cluster infrastructures with time to clean and restart the nodes

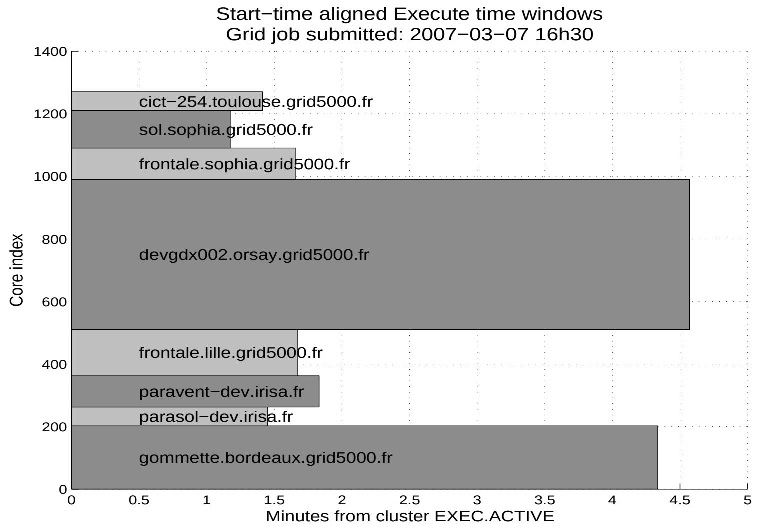

Figure 2.14: Cluster start-time aligned execution phase, showing overall (time-shifted) “Grid” time window for grid job execution. This shows wasted computing power as fast nodes sit idle waiting for slow nodes to complete.

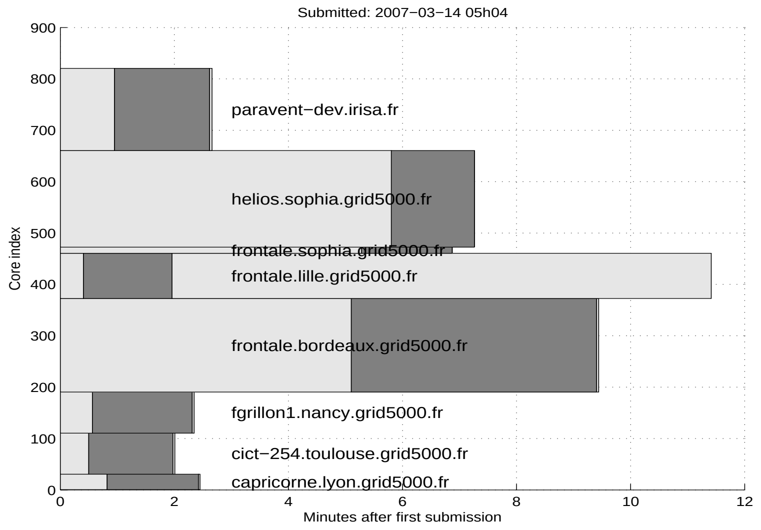

Figure 2.15: Even with reservations for multi-site synchronized parallel computing it is difficult to coordinate resource acquisition. Light grey boxes show waiting time, dark boxes show execution time.

after previous reservations ending at 6 AM were completed. In fact, this result does not include clusters which completely failed to respect the reservation or failed to initiate the job, and also three clusters with abnormal behaviour: two where the leader node failed to initiate the task on the workers, and one where the task commenced three hours late. It can be seen that only five clusters, consisting of approximately half of the 800 cores, were able to start the computation within a minute of the reservation. This poses serious problems for coordinated, pre-scheduled, pre-planned multi-cluster parallel computations, since it suggests it is difficult to have disparate clusters available during the same time window. The Lille cluster shows grey box for waiting time *after* execution, due to problems with synchronizing data from worker nodes back to leader node and user home directory. Other nodes also have this stage, but it takes $< 1s$ so is not visible at this scale (see *paravent-dev.inisa.fr*).

These studies have quantitatively revealed the difficulties with executing coordinated multi-cluster and multi-site parallel computational tasks. The layered state machine model from Section 2.7.1.2 for grid jobs has facilitated detailed tracking of state transitions through the various layers of the grid, and been a part of identifying mis-configured clusters and nodes. The metrics defined in Section 2.7.1.3 provide measures which suggest a $90 - 95\%$ occupancy efficiency at the cluster level is reasonable if the clusters are correctly configured and operating normally. Regarding parallel computing at the cluster level, it is clear that heterogeneity is irregular, even when a cluster claims it is composed of identical machines. Obviously, the heterogeneity is the essence of grid computing and it causes many challenges for the use of parallel computing on the grid.

2.7.2.2 Speedup, Load Balancing and Fault Tolerance

We have chosen an European option pricing using Monte Carlo methods to demonstrate the performance of a high performance computing system for financial computations. Because due to the "embarrassingly parallel" property of such pricing problem and the fact that, the simulated price can be verified through the Black-Scholes formula.

We developed a distributed pricing algorithm for the general high–dimension European option which is detailed in [18]. We describe the architecture and algorithmic principles, in the case of an European option on a basket of d assets, the other cases having a very similar architecture. When d is large ($d = 40$ in our experiment), basket option is particularly the most time consuming among European option types. The underlying prices is a vector $(S_i, i = 1, \ldots, d)$. One basket trajectory simulation consists in the simulation of the d prices (S_i) . Because of the correlation in the model, the trajectory simulation of the prices must be synchronized. A task sent to a worker (aka. processor) consists in asking it to simulate a number of simulations. The main problem regarding load balancing will be to fix the optimal value of package size according to the total number of simulations $nbMC$ and the dynamic load of the processors of the grid.

We have evaluated PicsouGrid with a one year maturity European put option on a basket of 40 correlated assets computed on the basis of 10^6 Monte Carlo trajectories (multidimensional Black-Sholes model with one step a day, 25% volatility per asset, 0.5% correlation, the precision is about 10^3). Experiments were run on Grid'5000 without introducing failure.

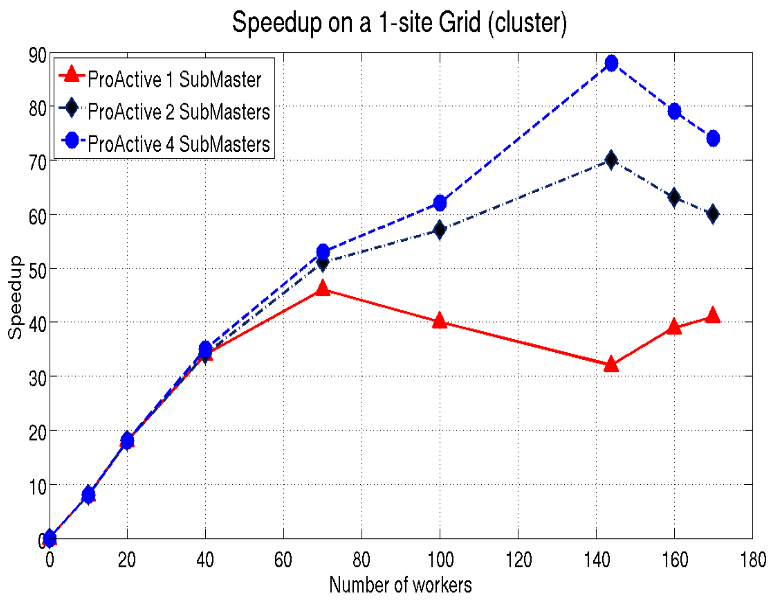

Figure 2.16: Speedup on a 1-site grid

According to the PicsouGrid architecture in Figure 2.2, it is relevant to adapt the number of sub-masters to the size of the grid, see Figure 2.16. In our experiment on one site, one sub-master is enough to manage up to 40 processors, then 2 sub-masters are better to manage around 70 processors, and then 4 sub-masters are desirable to manage more processors. Identifying the best configuration is strategic. Performances are better up to approximately 130 workers, and decrease beyond, see Figure 2.16.

We have deployed PicsouGrid on 4 sites using Proactive, with 1 sub-master and up to 23 workers per site. Performance results are presented in Figure 2.17. 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. Such issue is easy to understand because several sites offer more easily a large amount of processors, but communications take longer and limit

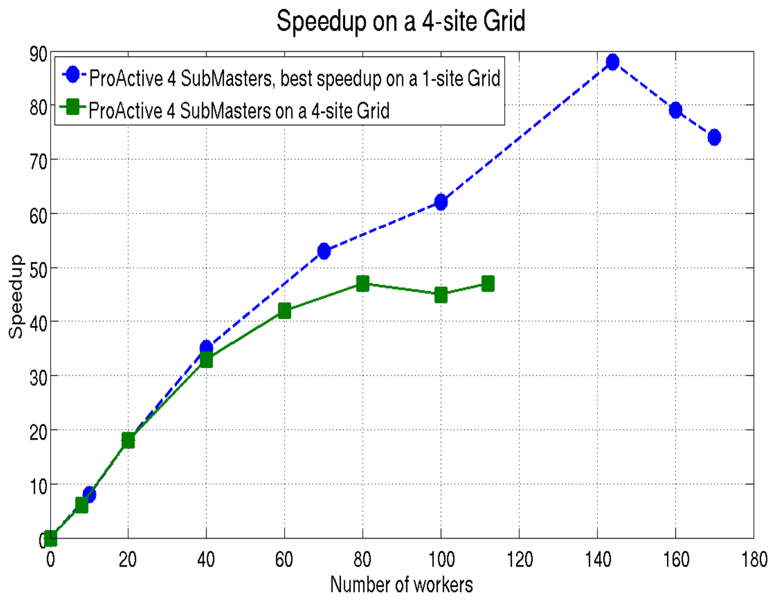

Figure 2.17: Speedup on a 4-site grid

the speedup.

We also performed PicsouGrid in case of failures and aggressive load balancing. As next performance results illustrate, when distributing independent Monte Carlo simulations, the heterogeneity and volatility of grid machines can be addressed by the aggressive task distribution introduced in Section 2.4, and defining an adequate trade-off for the task size.

Figure 2.18 shows the execution times of a basket pricing ($nbMC = 10^6$), function of the task size on a nominal grid of 144 processors ($nb - proc = 144$) and on the same grid on which we shutdowned 1 processor (a faulty grid of 143 processors). As no reserve processor is considered, the fault-resilience has to be entirely supported by the dynamic load balancing and the aggressive task distribution strategies. Small task size is adapted for highly volatile and heterogeneous grids as we can observe that in Figure 2.18. The failure impact is very limited for small task size, even if their numerous communications slow down computation. Optimal task size ($nbMC/144 = 6945$) on a sound grid 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, like 7000 is adapted to homogeneous and lightly volatile Grids: 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.

As a summary, we evaluated the performance of PicsouGrid by using a basket European

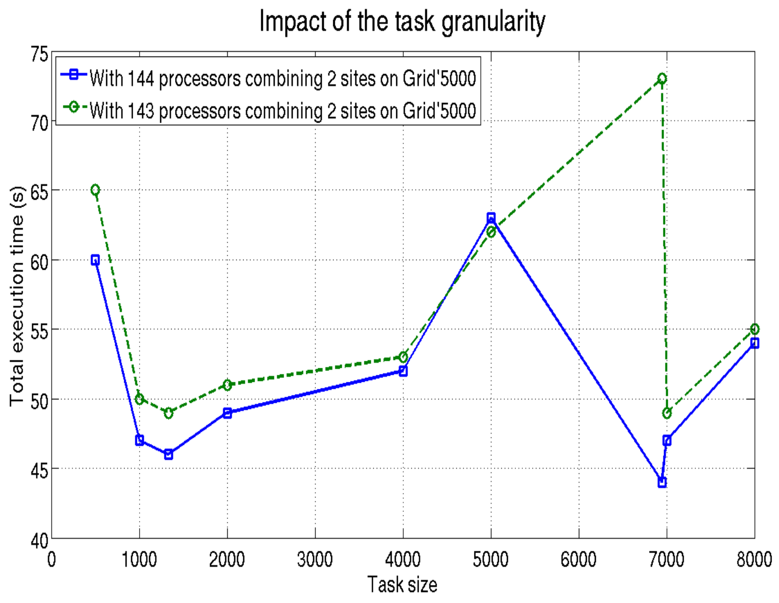

Figure 2.18: Impact of the task granularity

option pricing application. Such application is particularly the most time consuming among European option types which may require hours of computation on a single machine. According to the architecture in Figure 2.2, we varied the number of sub-masters to figure out the best configuration for PicsouGrid. It is shown that on a 1-site grid, one sub-master is enough to manage up to 40 processors, then 2 sub-masters for 70 processors and so on. We also experimented PicsouGrid on a multi-site grid (e.g. up to 4 sites) to figure out the slow down of speedup due to the communication overhead between sites. Overall we achieved good performances for basket European option pricing application on both one and multi-site grid. In term of task distribution, by creating a faulty grid, we observed that aggressive task distribution achieves good performances on sound and faulty Grids when computing independent Monte Carlo simulations.

2.8 Conclusion

In this chapter, we presented PicsouGrid a distributed computing framework in our financial context which consists of financial option pricing applications, to provide fault tolerance, load balancing, dynamic task distribution and deployment on a heterogeneous environment (e.g. a grid environment). PicsouGrid has been designed to support both large Bag-of-Tasks (BoT) problems and parallel algorithms requiring more communications. PicsouGrid has been successfully deployed on the multi-site grid (e.g. Grid'5000) for European option pricing. We achieved good performances for such application, using up to 190 processors on a cluster and up to 120 processors on a 4-sites grid. Fault-tolerance and load balancing are realized transparently for the programmer, based on processor replacement and dynamic and aggressive load balancing.

We also tried to figure out the latencies and behaviour within the various layers of the grid environment on which PicsouGrid applications were being run. Regarding experiments for such problem in Section 2.7.2, it is clear that heterogeneity of grid is important, even on a single cluster composed of identical machines. Latencies in binding grid tasks to particular nodes and initiation of tasks on a particular core can introduce delays of several seconds to minutes. This presents two major challenges for parallel computing on the grid: i) synchronization of the task start time; and ii) partitioning of the computational load. While static measures of relative performance on a cluster or node level are valuable, it is clear that these cannot always be trusted, hence it is reasonable to imagine the need for dynamic, application-layer determination of node performance prior to the initiation of parallel computations. Ideally this responsibility would be taken by the grid infrastructure (and by implication of the cluster owner), however the federated and unreliable nature of grids suggests the user or application needs to manage this for the present. At the level of multi-sites grid computing the key challenges are coordinated reservation and start-up. Our work has not investigated what granularity of computations are practical, however the start-up delays and unreliable fulfilment of reservations suggest that "contributory best-effort" workers may be appropriate, where workers enter and exit a worker pool in a dynamic fashion, and are acquired by a simulation manager and assigned to particular computations "on demand", rather than with a simulation manager expecting a set of workers based on a prior reservation.

Since the European option pricing can be expressed as an embarrassingly parallel problem, it is necessary to experiment PicsouGrid with more tightly-coupled applications, such as American pricing to evaluate the degree of parallelism which can be achieved, and at what cost, and to discover the performance impacts of real multi-cluster (multi-site) communicating parallel computations in a grid environment. In the next Chapter, we are going to introduce the American option pricing algorithms using Monte Carlo simulations. We will present a parallel approach for one of these algorithms and also evaluate the scalability of such proposed parallel approach in a computational grid environment using PicsouGrid.

Chapter 3

PARALLEL HIGH DIMENSIONAL AMERICAN OPTION PRICING

Pricing high dimensional American option is a hard and computational intensive problem in finance. This chapter focuses on the parallelization of the Classification Monte Carlo algorithm proposed by Picazo (2002) [94] for high dimensional American option pricing. First part of this research work has been accepted to Mathematics and Computers in Simulation Journal, Elsevier, 2010. Preliminary high dimensional American option pricing results were also published in the 1st *Workshop on High Performance Computational Finance in conjunction with the 2008 Supercomputing Conference, see [40].*

3.1 Motivation

According to the Eurex and ISE [1] annual report, in 2009 the equity derivatives segment of the Eurex derivatives exchanges board (options and single stock futures) reached 421 million contracts. Most of contracts are of the American type. Typical approaches for American option pricing are the binomial method of Cox and Rubinstein, see Cox and al. (1979) [37] or later in Cox and Rubinstein (1985) [38], the partial differential equation approach (see Wilmott and al. (1993) [126]) and Monte Carlo based methods (e.g. see Glasserman (2004) [52]). However, since binomial methods are not suitable for high dimensional options, MC simulations have become the cornerstone for simulation of American options in the industry. Hence in this thesis context, we only discuss about the use of Monte Carlo based methods for American option pricing. Such MC simulations have several advantages, including ease of implementation, applicability to multidimensional options and suitability to parallel and distributed computing. A litte survey showed that most of Monte Carlo based algorithms such as Andersen and Broadie (2004) [5], Kogan and Haugh (2004) [58], Meinshausen and Hambly (2004) [85], Rogers (2002) [99], Ibanez and Zapatero (2004) [63], Picazo (2002) [94] involve calculating an expectation, which itself can be resolved using another set of MC simulations. This "simulation on simulation" is computationally expensive. For example pricing American options with a large number of underlying assets is computationally intensive and might take several hours or even days (e.g. on a single processor). For instance, Ibanez and Zapatero (2004) [63] state that pricing the option with just five assets takes two days, which is not affordable in modern time critical financial markets.

In the literature, there exist some parallel American option pricing techniques. For example Huang (2005) [60] or Thulasiram (2002) [116], both implemented a parallel approach which is based on the binomial lattice model but are not suitable in case of high dimensional problems. Wan and al. (2006) [124] developed a parallel approach for pricing American options on multiple assets. Their parallel approach is based on the low discrepancy (LD) mesh method which combines the quasi-Monte Carlo technique with the stochastic mesh method. Muni Toke (2006) [92] investigated a parallel approach for the Monte Carlo based algorithm proposed by Ibanez and Zapatero (2004) [63]. In the recent studies, Bronstein and al. (2008) [27] presented a parallel implementation of the optimal quantization method to price multi-dimensional American options. Choudhury and al. (2008) [9] identified some opportunities for parallelization in the Least-Squares Monte Carlo (LSM) algorithm. Although the parallel efforts for the parallel approaches above have been made, only few of them reported using many CPUs at once (e.g. in [92] the experiments were performed on

[1] http://www.eurexchange.com

up to 12 processors). Moreover, in almost experiments the authors only evaluated their approaches only on small dimensional problems (e.g. maximum option on 5 assets [5, 63], minimum option on 10 assets [99, 94]). Thus there are no numerical results with dimension bigger than 10 that have been experimented in these research works and available in the literature.

Therefore, our research objective aims to overcome these limits described above by investigating a new parallel American option pricing algorithm. We provide a parallel approach for the Classification Monte Carlo (CMC) algorithm proposed by Picazo (2002) [94]. Such algorithm in its sequential form is similar to recursive programming so that at a given exercise opportunity it triggers some independent MC simulations to compute the continuation value. The optimal exercise strategy of an American option is to compare at each opportunity date the exercise value (intrinsic value) with the continuation value (the expected cash flow from continuing the option contract), and exercise if the exercise value is more valuable. The CMC algorithm classifies the underlying asset prices into two separated regions, the exercise region $\mathcal{S}$ and the continuation region $\mathcal{C}$. Using a classification technique borrowed from the machine learning domain, CMC algorithm calculates a model that characterizes the optimal exercise boundary separating these two regions for each opportunity date. Next, it estimates the option price using MC simulations based on such models. Our roadmap is to study the algorithm in detail to highlight its potential for parallelization: Overall we aim to increase the whole algorithm scalability, so for each of computation phases, our goal is to identify where and how the computation could be split into independent parallel tasks. That provides us a parallel approach so that it can be easy handled in by the computational grid. As we aim for a scalable enough approach, we should then be able to evaluate the American option pricing for high dimensional problems. So the main guideline of this chapter is to investigate the parallelization of the CMC algorithm to explore all the opportunities to reduce the pricing time by harnessing the computational power provided by the computational grid.

Chapter 3 is organized as follows: in Section 3.2 we provide an overview of use of Monte Carlo based methods for high dimensional American option pricing including regression based methods and parametric approximation methods. In Section 3.3 we will discuss in details relevant related works, for example some parallel approaches for the original LSM algorithm of Longstaff and Schwartz (2001) [77] and the original algorithm of Ibanez and Zapatero (2004) [63]. Based on these parallel approaches, we shift our interest on parallelization of another algorithm for high dimensional American option pricing, the CMC algorithm of Picazo (2002) [94] and provide a parallel approach for such algorithm in Section 3.4. We further detail in this section the use of two machine learning techniques that are Support Vector Machines and Booosting in the CMC algorithm. We figure out the impact of these two techniques on the overall accuracy and the parallel computational performance for the CMC algorithm. We perform some numerical results and analyze the performance speedup further in this section. As conclusion, in Section 3.5 we discuss about the advantages and limitations of our parallel approach and suggest some future research directions.

3.2 High Dimensional American Option Pricing

This section presents several Monte Carlo based methods that address the problem of high dimensional American option pricing and hedging. All of these methods require some substantial computational efforts and a large computational time.

3.2.1 Monte Carlo Based Methods

It is clear that pricing and hedging algorithms for American options are still challenging. Only few closed forms solutions have been found for some special cases. Particularly, pricing American options on a basket of underlying assets or with multiple risk factors is a complicated and difficult problem. For the last decade, many methods have been proposed but none of them are satisfactory. To date, Monte Carlo based methods act as the most promising methods to solve this problem. In fact, starting from the research of Tilley (1993) [117], people were encouraged to investigate the possibility of pricing American options using Monte Carlo methods. This and other early methods such as Barraquand and Martineau (1995) [15], Broadie (1997) [24], Fu and Hu (1995) [48] are reviewed in Boyle et al (1997) [21] and a comparison of methods is made in Fu et al. (2001) [49]. Many of these methods are also explained in Glasserman (2004) [52] and Tavella (2002) [113].

In Chapter 1, we briefly presented the American option pricing problem. In this section, we are going to detail two ways to formulate such problem: the Optimal Stopping Formulation and the Free Boundary Formulation. Based on these formulations, we also introduce two branches of Monte Carlo based methods for American option pricing: the Regression Based Methods and the Parametric Approximation Methods, see Figure 3.1.

3.2.1.1 Free Boundary Formulation - Parametric Approximation Methods

Since the American option can be exercised at any time up to the maturity date T, it is important to compute when the option should be exercised. During the option life T, at any time we have to determine the optimal exercise boundary

$$b^*(t) = \{\mathbf{x}; V(\mathbf{x}, t) = \Psi(\mathbf{x}, t)\}$$

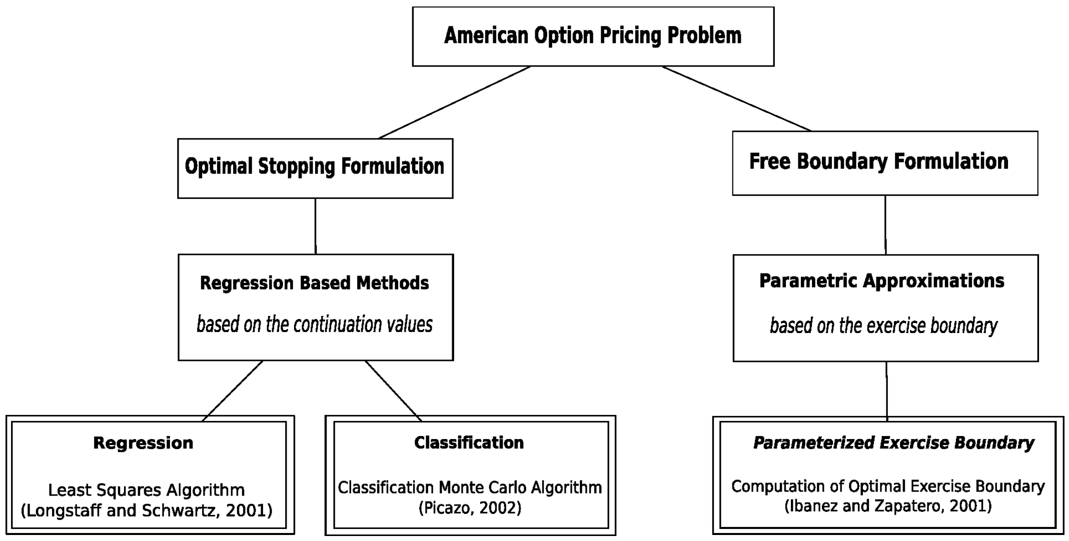
American Option Pricing Problem
Optimal Stopping Formulation
Free Boundary Formulation
Regression Based Methods
based on the continuation values
Parametric Approximations
based on the exercise boundary
Regression
Least Squares Algorithm
(Longstaff and Schwartz, 2001)
Classification
Classification Monte Carlo Algorithm
(Picazo, 2002)
Parameterized Exercise Boundary
Computation of Optimal Exercise Boundary
(Ibanez and Zapatero, 2001)

Figure 3.1: Monte Carlo based American Option Pricing Algorithm Family

where $V(\mathbf{x}, t)$ is the option price at time t and $\Psi(\mathbf{x}, t)$ is the payoff value at time t. However, the unknown form of the optimal exercise boundary is a major problem when using Monte Carlo simulation to price American options. Hence there exists a group of approaches for American option pricing which focuses into parameterizing the option exercise boundary, named parametric approximation methods. For the single asset cases, Ju (1998) [65] approximated the exercise boundary by a piecewise exponential curve and Ait Sahlia and Lai (2001) [69] did it with a four pieces linear spline. Such approaches gave a good approximation for the American option price. However, the structure of the exercise boundary in higher dimensions is not simple, as investigated by Broadie and Detemple (1997) [24], Villeneuve (1999) [123]. Later, Ibanez and Zapatero (2004) [63] used a parametric approach to obtain the optimal exercise boundary function for pricing American option. We will introduce this approach in the next paragraphs.

Ibanez and Zapatero (2004) [63] proposed an algorithm that computes explicitly the optimal exercise boundary. They consider a given finite set of opportunity dates $\Theta = \{t_m, m = 1, \ldots, N\}$. They provided an American option pricing algorithm based on the estimation of the optimal exercise boundary $b^*(t)$ at any opportunity date $t \in \Theta$. The algorithm of Ibanez and Zapatero is based on the observation that when the payoff Ψ is monotone and convex the optimal exercise boundary $b^*(t)$ relies on the set of points $\mathbf{x}^*$ that solve

$$V(\mathbf{x}^*, t_m) = \Psi(\mathbf{x}^*, t_m) \tag{3.1}$$

, see Broadie and Detemple (1997) [24].

Consider a basket American option on a basket of d assets $\overline{S}_t = \{S_t^i : i = 1, \dots, d\};\ t \in \Theta$. Denote B_t a $(d-1)$-dimensional vector of $d-1$ assets excluding S_t^i: $B_t = \left\{S_t^1, \dots, S_t^{i-1}, S_t^{i+1}, \dots, S_t^d\right\}$. In order to obtain B_t, the algorithm creates a lattice of points, called "Good Lattice Points" (GLPs) as the seeds that help to simulate the value of B_t, see [63] for the construction of such GLPs. Such lattice is of size $(nb\_GLP \times d-1)$, where nb_GLP is the number of points that will be used to form the boundary b^* and d is the number of assets. The optimal exercise boundary at time $t \in \Theta$ is a function of B_t, $F(B_t, t)$. Such function can be calculated recursively from t_{N-1} to t_1 because at t_N the function is simply the constant strike price K. Starting at $t_m, m = N-1$, take an initial point $\overline{S}_{t_m}^{1,(1)}$ (often it is the strike price K), the algorithm uses this point to compute the continuation value $C(\overline{S}_{t_m}^{1,(1)}, t_m)$ using Equation (3.4). Such expectation value can be estimated through straightforward MC simulations as in case of European option described in Equation (1.21). We note nb_cont the number of MC simulations for pricing such continuation value. Once having $C(\overline{S}_{t_m}^{1,(1)}, t_m)$, we can find $S_{t_m}^{1,(2)}$ that solves the following equation:

$$C(\overline{S}_{t_m}^{1,(1)}, t_m) = \Psi(\overline{S}_{t_m}^{1,(2)}, t_m) \tag{3.2}$$

Similary using $S_{t_m}^{1,(2)}$, we are going to find $S_{t_m}^{1,(3)}$ using (3.2) and so on until:

$$\left|S_{t_m}^{1,(n+1)} - S_{t_m}^{1,(n)}\right| \leq \epsilon \tag{3.3}$$

for a given small ϵ and $n \in \mathbb{N}^+$. Then we obtain an optimal point $S_{t_m}^{1,(*)} = S_{t_m}^{1,(n+1)}$. Such procedure will be repeated until we obtain the full set of optimal points $\{S_{t_m}^{j,(*)}; j = 1, \dots, nb\_GLP\}$ in order to be able to build the optimal exercise boundary. We refer reader to Ibanez and Zapatero (2004) [63] for the full explanation for the convergence condition of this step (e.g. choosing value of ϵ, number of iterations n). Once having the optimal points set, we can perform a regression on these points to obtain the boundary as a parametric curve (e.g. a second or third degree polynomial). Finally, once having the overall optimal exercise boundary at each opportunity, the option price at time $t \in \Theta$ can be estimated using Monte Carlo simulations as follows:

$$V(S_t, t) = \mathbb{E}\left[e^{-r(\tau - t)} \max\left(\Psi(\overline{S}_\tau, \tau) - K, 0\right)\right], \tag{3.4}$$

with $\tau = \min\{\tau > t, \tau \in \Theta; \overline{S}_\tau \geq \overline{S}_\tau^{(*)} = F(B_\tau, \tau)$ for a call or $\overline{S}_\tau \leq \overline{S}_\tau^{(*)}$ for a put$\}$. The following Algorthim 4 showes the pseudo-code of the Ibanez and Zapatero algorithm

The algorithm of Ibanez and Zapatero has some advantages as follows: Firstly it is able to provide the full detailed optimal exercise boundary to option holder and secondly

Algorithm 4 Serial Ibanez and Zapatero Algorithm

```
Require: $S_0^i$, $d$, $r$, $\delta_i$, $\sigma_i$, $T$, $N$,
Require: number of optimal boundary points $nb\_GLP$,
Require: number of trajectories to estimate each continuation value $nb\_cont$
Require: number of trajectories to estimate the final option price $nbMC$
 1: Generate the "Good Lattice Points", $B^j \in \mathbb{R}^{d-1}, j = 1, \ldots, nb\_GLP$
 2: [phase 1] :
 3: for $m = N - 1$ to 1 do
 4:    [step 1] : // Calculate $nb\_GLP$ optimal boundary points $S_{t_m}^{j,(*)} : j = 1, \ldots, nb\_GLP$.
 5:    for $j = 1$ to $nb\_GLP$ do
 6:       Compute the continuation value $C_{t_m}$ based on Equation (3.4) using $nb\_cont$ simulations
 7:       Calculate the $j^{th}$ optimal boundary point $S_{t_m}^{j,(*)}$ using Equations (3.2) and (3.3).
 8:    end for
 9:    [step 2] : Form $F(B_{t_m}, t_m)$ based on the set $\{B_{t_m}^j, S_{t_m}^{j,(*)} : j = 1, \ldots, nb\_GLP\}$
10: end for
11: [phase 2] : Generate new $nbMC$ paths $\{S_{t_m}^{i,s} : i = 1, \ldots, d; m = 1, \ldots, N; s = 1, \ldots, nbMC\}$. Using these calculated $F(B_{t_m}, t_m)$ above, we can estimate the final option price through Equation (3.4).
12: return the final option price.
```

it can be treated as an European option for both pricing and hedging at the final phase. However in term of computing performance, Ibanez and Zapatero stated in their paper that for pricing a maximum option of five assets, the algorithm took two days, which is not desirable in modern time critical financial markets. To overcome this limitation, Muni Toke (2006) [92] investigated a parallel approach fo Ibanez and Zapatero algorithm which will be further discussed in this chapter.

3.2.1.2 Optimal Stopping Formulation - Regression Based Methods

Such formulation is a common way to formulate the American option pricing problem. In Chapter 1, we presented the formula for a American option fair price at any time t before T using the optimal stopping formulation:

$$V(S_t, t) = \sup_{\tau \in \Pi(t,\tau)} \mathbb{E}\big[e^{-r\tau}\Psi(S_\tau, \tau \in [t, T])\big].$$

where S_t is the asset prices at time $t \in \Pi(t, T)$, with $\Pi(t, T)$ is the set of stochastic stopping time in $[t, T)$. This supremum is achieved by an optimal stopping time τ^* of the form:

$$\tau^* = \inf\{t \geq 0 : S_t \in \mathcal{S}(t)\}$$

where $\mathcal{S}(t) = \{\mathbf{x}; V(\mathbf{x}, t) \leq \Psi(\mathbf{x}, t)\}$. To compute numerically $V(S_t, t)$, we used the dynamic programming principle in Equation (1.8). For the convenience of the reader we recall here such Equation (1.8):

$$\begin{aligned} &V(S_{t_N}, t_N) = \Psi(S_{t_N}, t_N) \\ &V(S_{t_m}, t_m) = \max\Big\{\Psi(S_{t_m}, t_m), \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]\Big\} \end{aligned}$$

with the finite set of opportunity dates $\Theta = \{t_m, m = 1, \ldots, N\}$. Regarding the Equation (1.8), at a given opportunity $t_m \in \Theta$ and in a given state $S_{t_m} = \mathrm{x}$, we have $\Psi(S_{t_m}, t_m)$ as exercise value (also called intrinsic value) and the continuation value like:

$$C(\mathbf{x}, t_m) = \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m} = \mathrm{x}\big]. \tag{3.5}$$

Such expectation value can be estimated though straightforward MC simulations as in case of an European option described in Equation (1.21). Notice that $C(\mathbf{x}, T) \equiv 0$ and $C(\mathbf{x}, 0) = V(\mathbf{x}, t_m)$.

Using such dynamic programming representation, we can evaluate the option price $V(S_0, 0)$ by using a recursive backward computing. While the exercise value is easy to be computed, the estimation of the continuation value C is more challenging. There are several Monte Carlo based approaches for American option pricing essentially relying on the way how to estimate and use such continuation value defined in Equation (3.5). Regression based methods include Longstaff and Schwartz (2001) [77], Carriere (1996) [30] and Tsitsiklis and Van Roy, 2001 [119]. The method of Longstaff and Schwartz has been particularly popular and has been investigated by a number of authors such as Moreno and Navas (2003) [91], Stentoft (2004) [108]. Proof of convergence is given in Clement et al. (2001) [34]. Chaudhary (2005) [32] and Lemieux (2004) [76] have applied quasi-random sequences to the valuation of American options. Such algorithms above form a group of approaches for American option pricing, named Regression Based Methods.

The key of such family of approaches is that the continuation value $C(S_{t_m}, t_m)$ in (3.5) can be approximated as follows:

$$C(S_{t_m}, t_m) = \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m} = \mathrm{x}\big] = \sum_{r=1}^{M} \beta^r_{t_m}\psi^r(S_{t_m}) \tag{3.6}$$

for a set of basic functions $\psi^r(S_{t_m})$ and constant coefficients $\beta^r_{t_m}$. Using this continuation value formulation, the authors in Longstaff and Schwartz (2001) [77], Carriere (1996) [30] and Tsitsiklis and Van Roy, 2001 [119] have proposed the use of regression to estimate the continuation values from simulated paths then to compute the American option price by using MC simulations. For example, the Least Squares Monte Carlo (LSM) algorithm of Longstaff and Schwartz used the least squares regression to estimate the best coefficients β in the approximation (3.6). We refer the readers to Longstaff and Schwartz (2001) [77] for the full explanation of the LSM algorithm and to Clément, Lamberton and Protter (2001) [34] for the proofs of convergence of this algorithm. Algorithm 5 briefly presents the pseudo-code of the serial LSM algorithm.

Algorithm 5 Serial Original Least Square Monte Carlo Algorithm

```
Require: $S_0^i$, $d$, $r$, $\delta_i$, $\sigma_i$, $T$, $N$
Require: number of simulations $nbMC$
 1: [phase 1] : // Simulate $\{S_T^{i,s} : i = 1, \ldots, d; s = 1, \ldots, nbMC\}$
 2: for $m = N - 1$ to 1 do
 3:    [step 1] Firstly, realize the optimal stopping time at time $t_{m+1}$. Secondly, generate
       $nbMC$ paths of $\{S_{t_m}^{i,(s)} : i = 1, \ldots, d; s = 1, \ldots, nbMC\}$ asset prices from $t_{m+1}$.
 4:    [step 2] At $t_m$, estimate the best coefficients $\beta_{t_m}$ for $\sum_{r=1}^{M} \beta_{t_m}^r \psi^r(S_{t_m})$ using least
       square regression
 5:    for $s = 1$ to $nbMC$ do
 6:       Compute the continuation value for each path $C^{(s)}(S_{t_m}, t_m)$ using the estimated
          coefficients $\beta_{t_m}$
 7:       Compute the exercise value for each path $\Psi^{(s)}(S_{t_m}, t_m)$
 8:       if $C^{(s)}(S_{t_m}, t_m) \leq \Psi^{(s)}(S_{t_m}, t_m)$ then
 9:          set the optimal stopping time at $t_m$
10:       else
11:          set the stopping rule at $t_m$.
12:       end if
13:    end for
14: end for
15: [phase 2] Using the backward generated $nbMC$ paths $\{S_{t_m}^{i,(s)} : i = 1, \ldots, d; s =
    1, \ldots, nbMC\}; m = 1, \ldots, N$ and the optimal stopping time set above, we can estimate
    the final option price.
16: return the final option price.
```

Later, Picazo (2002) [94] stated that the approximation of the continuation value in regression based methods is not necessary. Based on dynamic programming representation in Equations (1.8), it is enough to decide between two ways, exercise, or not, and this does not require full knowledge of the functional form of the continuation value. Instead of using the continuation value formulation in Equation (3.6), Picazo focuses on Equation (1.8), classify the exercise values and continuation values into two regions $\mathcal{S}(t)$ and $\mathcal{C}(t)$ and uses a classification algorithm, a technique borrowed from machine learning, to characterize the boundary separating these two regions. Picazo named his algorithm as Classification

Monte Carlo (CMC) algorithm. Algorithm 6 briefly presents the pseudo-code of the serial CMC algorithm.

Algorithm 6 Serial Classification and Monte Carlo Algorithm

```
Require: $S_0^i$, $d$, $r$, $\delta_i$, $\sigma_i$, $T$, $N$.
Require: number of classification points $nb\_class$,
Require: number of trajectories to estimate each continuation value $nb\_cont$
Require: number of trajectories to estimate the final option price $nbMC$
 1: [phase 1] :
 2: for $m = N - 1$ to 1 do
 3:     Generate $nb\_class$ points of $\{S_{t_m}^{i,(s)} : i = 1, \ldots, d; s = 1, \ldots, nb\_class\}$.
 4:     [step 1] :
 5:     for $s = 1$ to $nb\_class$ do
 6:         Compute $C^{(s)}(S_{t_m}, t_m) = \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]$ using $nb\_cont$ trajectories and also compute $\Psi^{(s)}(S_{t_m}, t_m)$.
 7:         if $C^{(s)}(S_{t_m}, t_m) \leq \Psi^{(s)}(S_{t_m}, t_m)$ then
 8:             sign = 1
 9:         else
10:             sign = -1
11:         end if
12:     end for
13:     [step 2] : Classify $\left\{\left(S_{t_m}, \text{sign}\right)^{(s)} : s = 1, \ldots, nb\_class\right\}$ to characterize the exercise boundary at $t_m$.
14: end for
15: [phase 2] : Generate new $nbMC$ trajectories $\{S_{t_m}^{i,(s)} : i = 1, \ldots, d; m = 1, \ldots, N; s = 1, \ldots, nbMC\}$. Using the characterization of the exercise boundary above, we can estimate the final option price.
16: return the final option price.
```

The algorithms of both regression based and parameterized approximation methods can be divided into two different phases of computation. Currently, research works of Choudhury and al. (2008) [9], Abbas-Turki (2009) [1] figured out the important factors for parallelizing the LSM algorithm. While Choudhury and al. (2008) [9] deployed their parallel approach on a cluster of CPUs, Abbas-Turki (2009) [1] explored the use of a cluster of GPUs. However, there are no related research works for parallelizing CMC algorithm. We aim to provide a parallel approach for the CMC algorithm of Picazo (2002) [94] as one of the main contributions of this thesis.

3.3 Parallelization Strategies

3.3.1 Overview

In this section, we are going to summarize the parallel approaches for Ibanez and Zapatero algorithm proposed by Muni Toke (2006) [92] and for LSM algorithm proposed by Choudhury and al. (2008) [9] and Abbas-Turki (2009) [1]. Our parallel approach for CMC algorithm will be fully described further. The overview of these parallel approaches are described in Figure 3.2. As observed, the three original algorithms (Ibanez and Zapatero, LSM and CMC) could be separated into two different phases of computations. So the authors listed above, investigated the opportunity of parallelism for each phase.

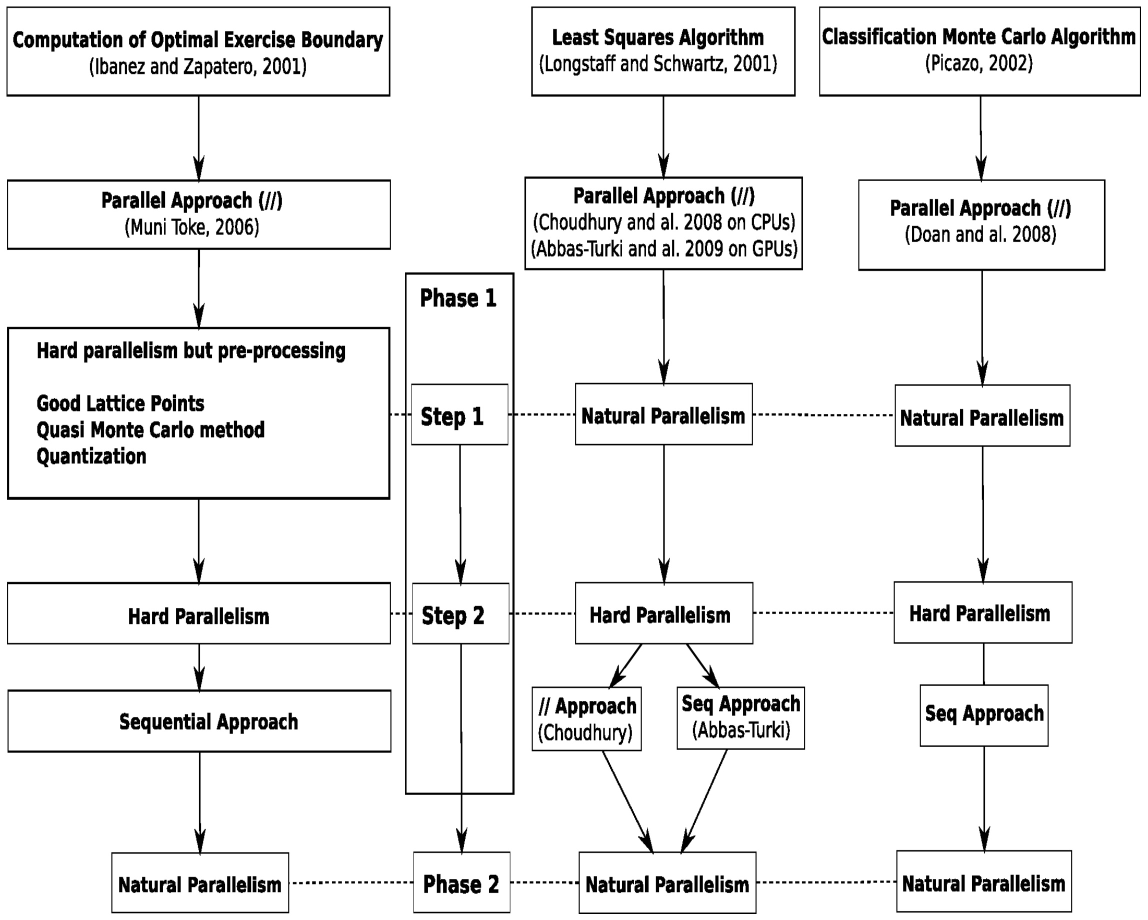

Figure 3.2: Parallel strategies for Monte Carlo based methods

Of course, the source of parallelism within each phase is not the same. The **[phase 1]** of the LSM and CMC algorithm consists of the use of Monte Carlo methods to generate the path of asset prices which is naturally parallelisable. Meanwhile the one of Ibanez and Zapatero algorithm uses a technique of Quasi Monte Carlo methods to create a lattice of "Good Lattice Points" and then it bases on these points to calculate the optimal boundary points. Since the lattice is pre-calculated, Muni Toke (2006) [92] divided such lattice into independent sub-lattices for being suited with parallel computing. There is one common point, as we can observe from the Algorithms described above: the [**phase 1**] of all three algorithms includes two sub-steps of computation. For example, in case of LSM algorithm the [**phase 1**] includes the [**step 1**] that realizes the optimal stopping time rule and [**step 2**] that estimates the continuation values using the least square regression. Meanwhile in case of Ibanez Zapatero algorithm, the [**phase 1**] includes the [**step 1**] which computes the continuation values in order to estimate the optimal exercise boundary points and the [**step 2**] which performs the boundary based on these optimal points. Similarly as in case of Ibanez and Zapatero algorithm, the [**phase 1**] of CMC algorithm contains 2 steps. The first [**step 1**] aims to compute the continuation values and the exercise value as well to decide to continue or exercise the option contract. The [**step 2**] performs a classification step in order to build a classifier model which can help the option holder to get the right decision.

However, while the [**step 1**] can be parallelized, the [**step 2**] of the three algorithms is the most difficult step to be parallelized. In fact in parallelizing the Ibanez and Zapatero algorithm, Muni Toke (2006) [92] kept running this [**step 2**] sequentially. In case of LSM algorithm, Choudhury and al. (2008) [9] provided a parallel approach for such step on a cluster of CPUs environment. In the parallel approach of Abbas-Turki (2009) [1] for LSM algorthim using a cluster of GPUs, this [**step 2**] is kept running sequentially.

For the CMC algorithm, we investigate the parallel approach for the [**phase 1**] by parallelizing the [**step 1**] and keep the [**step 2**] serial. We also discuss a bit about the parallel approach for the [**step 2**] in our perspective section. Finally, the last [**phase 2**] in fact relies on straightforward Monte Carlo simulations like in case of an European option. Hence from now on, we will not discuss on the parallelism for this phase anymore.

3.3.2 Parallel Approach for Ibanez and Zapatero Algorithm

In his work [92], Muni Toke stated that the computation of *nb_GLP* optimal boundary points of [**step 1**] can be simulated independently, see Algorithm 4. The [**step 1**] will return the optimal boundary points for the computation of the optimal exercise boundary in [**step 2**]. Finally, the option price can be estimated using straightforward MC simulations in [**phase 2**] which is easy to be parallelized. Muni Toke was successful to deploy his parallel approach on a homogeneous parallel cluster using up to 12 processors.

In order to deploy it in a more heterogeneous environment like a grid, we implemented such approach using PicsouGrid. We also investigated the sensitivities of estimated option prices with the change of some numerical parameters such as the number of optimal boundary points *nb_GLP*, the number of simulations to compute the continuation value *nb_cont*. Muni Toke (2006) [92] noted that *nb_GLP* smaller than 128 is not sufficient and prejudices the option price. In order to observe the effect of *nb_GLP* on the accuracy of the estimated price, we experimented the algorithm with varied number of *nb_GLP* starting from 128 points, see Table 3.1. For these experiments, we consider a call American option on the maximum of three assets. The asset prices follow the BSE (1.17). The call payoff at time t is defined as $\Psi(S_t, t) = (\max_{i=1,..,d}(S_t^i) - K)^+, d = 3$. The underlying assets parameters are described as follows:

$$\begin{aligned}&S_0^i = 90, K = 100, r = 0.05, \delta = 0.1, \sigma = 0.2 \text{ and maturity date } T = 3 \text{ years}\\ &\text{number of exercise dates } N = 9, \epsilon = 0.01\end{aligned} \tag{3.7}$$

While keeping the number of trajectories for continuation value computation $nb\_cont = 5000$, the results in Table 3.1 indicate that increasing number of *nb_GLP* has very little impact on the accuracy of the estimated price compared to the reference price. The reference price of such option is taken from Andersen and Broadie (2004) [5]. First, the authors reported the price of 12.90 which was determined from the multidimensional BEG routine of Boyle et al. (1989) [22] (in the case of 3 assets, the BEG results were from 270 time steps with an approximate error of 0.015). Second, Andersen and Broadie (2004) [5] also computed the lower and upper bound for this option price using their primal dual simulation algorithm. They obtained the lower bound of 11.279 ± 0.007, based on the LSM algorithm of Longstaff and Schwartz with 2×10^6 Monte Carlo simulations. They computed the upper bound which is 11.290 ± 0.009 using extra simulations to estimate the martingale in the stopping (exercise) region $\mathcal{S}(t)$. Notice that both lower and upper bound of the American option was computed with only 9 opportunity dates. In general, we consider the price of 11.290 is the American price of this maximum call option on 3 assets. However, we observe a linear increase in the computational time when increasing the number of *nb_GLP*, interesting information to set up a trade-off between the computational time and the accuracy of results.

Now let us focus on the effect of *nb_cont*, the number of simulations required to compute the continuation value. In [92], the author commented that the large values of *nb_cont* do not affect the accuracy of option price. For these experiments, we set $nb\_GLP = 128$ and vary *nb_cont* as shown in Table 3.2. We can clearly observe that *nb_cont* in fact has a strong

nb_GLP	Price	Time (in minute)	Abs Error	Relative Error
128	11.254	4.6	0.036	0.31%
256	11.258	8.1	0.032	0.28%
1024	11.263	29.5	0.027	0.24%

Table 3.1: Impact of the value of *nb_GLP* on the results of the maximum on three assets option ($S_0 = 90$) while keeping *nb_cont* = 5000 The reference price is 11.290. Running time is reported based on 16 processors.

nb_cont	Price	Time (in minute)	Abs Error	Relative Error
5000	11.254	4.6	0.036	0.31%
10000	11.262	6.9	0.028	0.24%
100000	11.276	35.7	0.014	0.12%

Table 3.2: Impact of the value of *nb_cont* on the results of the maximum on three assets option ($S_0^i = 90$). Running time on 16 processors.

impact on the accuracy of the computed option prices: the computational error decreases with the increased *nb_cont*. A large value of *nb_cont* results in more accurate boundary points, hence more accurate exercise boundary. Further, if the exercise boundary is accurately computed, the resulting option prices are much closer to the true price. However this, as we can see in the third column, comes at a cost of increased computational time.

The results obtained clearly indicate that the scalability of Ibanez and Zapatero algorithm is limited by the boundary points computation. The Table 3.1 showed that there is no effective advantage to increase the number of such points as will, just to take advantage of a greater number of available CPUs. Moreover, the time required for computing individual boundary points varies and the points with slower convergence rate often haul the performance of the algorithm.

3.3.3 Parallel Approach for Least Square Monte Carlo Algorithm

Choudhury and al. (2008) [9] explored the parallelization of the LSM algorithm proposed by Longstaff and Schwartz (2001) [77]. The parallel approach separated the LSM algorithm into three blocks including path simulation, regression/set stopping time and pricing which correspond to **[step 1]** and **[step 2]** in the **[phase 1]** and **[phase 2]** in Algorithm 5 respectively. The main technique used in the **[step 1]** and **[phase 2]** is Monte Carlo methods, which are known to be easily parallelized and ideally scalable. Hence the author focused on the challenge of parallelizing the **[step 2]**, full explanation can be found in [9]. However in [9], the authors only implemented their parallel approach in the case of one dimensional American option pricing.

The authors implemented their parallel approach using QuantLib library [4] and experimented the implementation on a Blue Gene/P system. The number of processors was varied from 4 to 32. Since the objective of the experimentation is not scaling the whole algorithm to a very large number of processors, they put their interest into analyzing the gained performance for each phase while varying the parameters. The performance results showed that the authors gained a good speedup for both path simulation and pricing phases due to the natural scalability of the Monte Carlo simulations. The most computation intensive is the regression/set stopping time step which depends on the strike price, volatility and the choice of regression basis function. In order to analyze this step performance, the authors considered two scenarios: in the first case they fixed the basis function and vary the strike price and volatility, while in the second one, they fix the last two parameters and vary the basis function. The performance results for the regression/set stopping time step showed that such step scaling is better for the monomial basis function in comparison to other basis functions (e.g. polynomial). The reason is that there still exists a bottleneck in price updating.

The overall speedup is limited by the regression/set stopping time step timing. In term of numerical results, the authors did not provide an implementation in case of high dimensional American options which require much more intensive computing but which are closed to the real-life option trading. The main difficulty is that the complexity of regression step increases linearly with the number of dimensions.

3.4 Parallel Approach for Classification Monte Carlo Algorithm

In this section we will present our contribution in providing a parallel approach for the Classification Monte Carlo algorithm. We also investigate the use of several classification techniques for the **[step 2]** in CMC algorithm. Then, the section continues with some numerical results that were used in order to validate the CMC algorithm implementation. The parallel CMC algorithm was successfully experimented in both grid and cluster environments using PicsouGrid based upon the ProActive Parallel Suite [11].

3.4.1 Classification Problem

Consider a basket American option of d assets that can only be exercised at a finite set of opportunity dates $\Theta = \{t_m, m = 1, \dots, N\}$. From the Equation (1.8) we can see that at time t_m the option holder should exercise the option whenever $\Psi(S_{t_m}, t_m) >$

$\mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]$ and to hold it otherwise. Let us define

$$\phi(S_{t_m}, t_m) = \Psi(S_{t_m}, t_m) - \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big] \tag{3.8}$$

then the exercise region is described by $\mathcal{S}(t_m)) = \{x\big|\phi(x, t_m) > 0\}$ and continuation region $\mathcal{C}(t_m) = \{x\big|\phi(x, t_m) < 0\}$. It is enough to find a function $F(x, t_m)$ such that

$$\text{sign}F(x, t_m) = \text{sign}(\phi(x, t_m)) \tag{3.9}$$

Hence this function $F(x, t_m)$ consists of a characterization of the exercise boundary at time $t_m \in \Theta$ such that the region $\mathcal{S}(t_m) = \{x\big|F(x, t_m) \geq 0\}$ and the region $\mathcal{C}(t_m) = \{x\big|F(x, t_m) \leq 0\}$. We will further discuss about this characterization problem in the next section.

Finally, once having the characterization of the exercise boundary for $\forall t_m \in \Theta$ then we can estimate the option price using:

$$V(S_0, 0) \simeq \sup_{\tau\in[0,T]} \mathbb{E}\big[e^{-r\tau}\Psi(S_\tau, \tau \in [0, T])\big]. \tag{3.10}$$

where the supremum is achieved by an optimal stopping time $\tau^* = \min\{t_m \in \Theta\big|F(S_{t_m}, t_m) \geq 0\}$.

3.4.1.1 Characterizing Optimal Exercise Boundary Through a Classification Problem

In the American option pricing problem, there are only two regions $\mathcal{S}$ and $\mathcal{C}$. So the exercise region $\mathcal{S}$ can be labelled as "1" and the continuation one $\mathcal{C}$ as "-1". From (3.8), we are interested in characterizing the region where $\phi(x, t)$ is positive (class 1) as well as the region where it is negative (class -1). This problem in fact is a binary classification problem. Given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^{nb\_class}$, each $\mathbf{x}$ has d coordinates, $\mathbf{x} = (x^1, x^2, \ldots, x^d) \in \mathcal{R}^d$. Each point $\mathbf{x}$ belongs to a class $y_i \in \{-1, 1\}$. The objective is to build a classifier model using such dataset which is able to predict the class value for any new given point. There are a number of methods that address to solve the classification problem such as the boosting algorithms (for short boosting) in Schapire (1990) [102] and later in Freund (1996) [45] or the well-known kernel based Support Vector Machines (SVMs) in Cristianini and Shawe-Taylor (2000) [39].

Boosting is a general strategy for learning the final strong classifier for a given data by iteratively learning weak classifiers on such data. A decision tree is usually used as the weak classifier. Some of popular boosting algorithms are AdaBoost in Freund and Schapire (1996) [45], Gradient Boost in Friedman (2001) [47], etc. Meanwhile, the support vector machines are a group of supervised learning algorithms that can be applied for both classification and regression problems. More precisely, a support vector machine algorithm constructs a hyperplane in a high dimensional space that separates the data with different class labels.

In the next paragraphs, we will introduce 2 boosting algorithms and the support vector machine algorithm using kernel function, for our particular option pricing problem.

Gradient Logit Boost : In the original work of Picazo (2002) [94], the author provided a boosting algorithm based on a logit loss function, named *cdb-logitBoost*, where *cdb* means characterization of decision boundary. Such proposed algorithm was based on the original algorithms Gradient Boost of Friedman (2001) [47] and AnyBoost of Mason and al. (1999) [84]. In fact, these two boosting algorithms address the regression problem where the main idea is to define a loss function then find its minimizer within a given class of functions. Picazo used a logit loss function in his own algorthim *cdb-logitBoost*. The criteria for choosing such loss function are:

- first, its minimizer should characterize the boundary.
- second, once the sign of the function F has ready been correctly obtained, it can stop the minimizing process.
- and third, it should penalize more heavily the mistakes while estimating the sign of F.

The logit loss function is described as follows:

$$\text{Loss}(y, F) = \log(1 + e^{-yF}),\ y \in \{-1, 1\}. \tag{3.11}$$

where $F(\mathbf{x}) = \log\left[\frac{Pr(y=1|\mathbf{x})}{Pr(y=-1|\mathbf{x})}\right]$, see the discussion in classification problem in Friedman and al. (2000) [46].

The pseudo-code of the *cdb-logitBoost* is described in the following Algorithm 7. For the full explanation of the *cdb-logitBoost* algorithm, see Picazo (2002) [94].

The weak learner mentioned at each iteration k in Algorithm 7 is a a decision tree with L terminal nodes which has the form as follows

$$f_k(\mathbf{x}) = \sum_{l=1}^{L} c_l^k \, \mathbb{1}\left(\mathbf{x} \in R_l\right) \tag{3.12}$$

One of the most popular method for building such decision tree is $CART^{TM}$ [107]. Given a set of points $\{(\mathbf{x}_i, y_i)\}_{i=1}^{nb\_class}$ with $(\mathbf{x}_i = (x_i^1, \dots, x_i^d) \in \mathcal{R}^d$ and $y_i \in \mathcal{R}$. $CART^{TM}$ constructs

Algorithm 7 Gradient Logit Boosting Algorithm

```
1: Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{nb\_class}$ and $F_0(x) = 0$.
2: for $k = 1$ to $K$ do
3:     Compute the pseudo response $\widehat{y}_i = y_i/(1 + e^{y_i F_{k-1}(\mathbf{x})})$
4:     Fit the weak classifier $f_k = \{R_l^{(k)}\}_{l=1}^{L}$, a decision tree with $L$ terminal nodes that
       minimizes $\sum_{i=1}^{N}(\widehat{y}_i - f(\mathbf{x}_i))^2$
5:     Let $c_l^{(k)} = \sum_{x \in R_l^{(k)}} \widehat{y}_i / \sum_{x \in R_l^{(k)}} \widehat{y}_i(y_i - \widehat{y}_i)$ for $l = 1, \ldots, L$.
6:     Let $F_k(\mathbf{x}) = F_{k-1}(\mathbf{x}) + \sum_{l=1}^{L} c_l^{(k)} \mathbb{1}(\mathbf{x} \in R_l^{(k)})$.
7: end for
8: Output the classifier $F_K(\mathbf{x}) = \sum_{k=1}^{K} \sum_{l=1}^{L} c_l^{(k)} \mathbb{1}(\mathbf{x} \in R_l^{(k)})$.
```

the tree as follows:

Each terminal node is defined by three parameters: the split variable, the split point and the constant c. First we define the left terminal node as $R_{\text{left}}(j,s) = \{\mathbf{x}|x^j < s\}$ and the right one as $R_{\text{right}}(j,s) = \{\mathbf{x}|x^j > s\}$, where j is the splitting variable ($j = 1, \ldots, d$) and $s \in \mathcal{R}$ is the splitting point. $CART^{TM}$ solves the following optimization problem to obtain j, s and c:

$$\min_{j,s}\Big(\min_{c_1\in\mathcal{R}} \sum_{\mathbf{x}_i \in R_{\text{left}}(j,s)} (y_i - c_1)^2 + \min_{c_2\in\mathcal{R}} \sum_{\mathbf{x}_i \in R_{\text{right}}(j,s)} (y_i - c_2)^2\Big) \tag{3.13}$$

with the constraint that both terminal nodes are nonempty and the optimal values of c_1 and c_2 are $c_1^* = \text{average}(y_i|\mathbf{x}_i \in R_{\text{left}})$ and $c_2^* = \text{average}(y_i|\mathbf{x}_i \in R_{\text{right}})$ with any value of j and s. Hence the Equation (3.13) can be reduced to

$$\min_{j,s}\Big(\sum_{\mathbf{x}_i \in R_{\text{left}}(j,s)} (y_i - c_1^*)^2 + \sum_{\mathbf{x}_i \in R_{\text{right}}(j,s)} (y_i - c_2^*)^2\Big) \tag{3.14}$$

the optimal values j and s can be found by considering all the possible split points for each variable.

Once having the $F_K(\mathbf{x})$, we can invert it to give the probability value such that

$$\begin{aligned} P(y = 1|\mathbf{x}) &= 1/(1 + e^{-F_K(\mathbf{x})}) \\ P(y = -1|\mathbf{x}) &= 1/(1 + e^{F_K(\mathbf{x})}) \end{aligned} \tag{3.15}$$

to predict the label of any given data point. We implemented this Gradient Logit Boost algorithm in PicsouGrid using Weka, a suite of machine learning software written in Java, developed at the University of Waikato [56].

Beside the Gradient Logit Boost mentioned above, we are also interested in performing the classification in our option pricing problem with other algorithms such as the AdaBoost algorithm of Freund (1995) [45] and a Support Vector Machines algorithm proposed by Platt (1999) [95]. These two algorithms are widely used to solve such classification problems and more important they provide some proportional opportunities for parallelism. In the following sections, we are going to detail both AdaBoost amd SVMs algorithms to figure out the motivation of using them in our particular American option pricing problem.

AdaBoost : In this paragraph, we present the AdaBoost (Adaptive Boosting) algorithm and how it can be applied for the option pricing. AdaBoost aims to fit a decision tree (classifier) to different reweighted versions of data then take a weighted majority vote of them to produce the final classification. We repeat the reweighted process in a number of iterations K, which is predetermined by the user, until we have the desired final function. In binary classification which is the case in our problem of option pricing, there are only two used classes label “1” and “-1”. The boosting procedure tries to find tree type model $F(\cdot)$ that classifies a given input data $\mathbf{x}_i$ to one of the two labels according to the $\operatorname{sign}(F(\cdot))$. Algorithm 8 shows how to estimate this function.

Algorithm 8 AdaBoost

```
1: Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{nb\_class}$, starting weights $w_i = 1/nb\_class$, $i = 1, .., nb\_class$ and $F_0(x) = 0$.
2: for $k = 1$ to $K$ do
3:     Fit the weak classifier $f_k(\mathbf{x}) \in \{-1, 1\}$ using weights $w_i$ (the classifier is usually a tree)
4:     Compute $e^{(k)} = \frac{1}{N_1} \sum_{i=1}^{N} w_i \, \mathbb{1}\,(y_i \neq f_k(\mathbf{x}_i))$ and $c^{(k)} = \log((1 - e^{(k)})/e^{(k)})$.
5:     Update weights $w_i \longleftarrow w_i \exp(c^{(k)} \; \mathbb{1} \; (y_i \neq f_k(\mathbf{x}_i)))$ and renormalize such that $\sum_{i=1}^{N} w_i = 1$.
6:     Let $F_k(\mathbf{x}) = F_{k-1}(\mathbf{x}) + c^{(k)} f_k(\mathbf{x})$.
7: end for
8: Output the classifier $\operatorname{sign}(F_K(\mathbf{x})) = \operatorname{sign}\left(\sum_{k=1}^{K} c^{(k)} f_k(\mathbf{x})\right)$.
```

In our experimentation, we consider the “decision stump” as the weak classifier. The “decision stump” is a special case of decision tree, which has only two terminal nodes and has the form:

$$f(\mathbf{x}) = y \, \mathbb{1}\,(x_j > s) \tag{3.16}$$

where $s \in \mathbb{R}$, $j \in \{1, \dots, J\}$ with J is the dimension of $\mathbf{x}$ and $y \in \{-1, 1\}$.

In fact, the AdaBoost algorithm can be viewed as the optimization of the exponential loss function, see Friedman and al. (2000) [46] for full explanation. Such loss function is described as follows:

$$\text{Loss}(y, F) = e^{-yF} \tag{3.17}$$

Support Vector Machines (SVMs) : SVMs are well–known data mining methods for classification and regression problems in machine learning. In this paragraph we describe how we can use SVMs for classifying the asset values. First, we will briefly discuss the mathematical background. Consider the dataset mentioned above $\{\mathbf{x}_i, y_i\}_{i=1}^{nb\_class}$, SVMs aims to find the maximum-margin hyperplane that divides the negative class points from those with positive class. Such hyperplane can be written as the set of points $\mathbf{x}$ satisfying

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{3.18}$$

where the vector $\mathbf{w}$ is a "normal" vector which is perpendicular to the hyperplane. The distance between the origin and the hyperplane in Equation (3.18) is $\dfrac{|b|}{\|\mathbf{w}\|}$ with $\|\mathbf{w}\|$ is the norm of the vector $\mathbf{w}$. Consider the points from the negative and the positive class that satisfy the following hyperplanes respectively

$$\mathbf{w} \cdot \mathbf{x} + b = -1 \tag{3.19}$$

and

$$\mathbf{w} \cdot \mathbf{x} + b = 1. \tag{3.20}$$

The distance between the origin and these two hyperplanes are $\frac{|-1-b|}{\|\mathbf{w}\|}$ and $\frac{|1-b|}{\|\mathbf{w}\|}$. Hence we have the distance between the hyperplanes (3.19) and (3.20) is $\frac{2}{\|\mathbf{w}\|}$. Now the problem of finding the maximum-margin hyperplane is equivalent to maximizing $\frac{2}{\|\mathbf{w}\|}$ or minimizing $\frac{\|\mathbf{w}\|^2}{2}$. The problem is formulated as follow

$$\begin{array}{l} \text{minimizing } \frac{\|\mathbf{w}\|^2}{2} \\ \text{with the constraints } y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1 \geq 0,\ i = 1, \ldots, nb\_class \end{array} \tag{3.21}$$

The optimization problem in Equation (3.21) represents the minimization of a quadratic function under linear constraints. A common method to solve such minimization problem in Equation (3.21) is using a Lagrange function, see [103] for more details. Based on the used of a Lagrange function, Equation (3.21) is transformed into its dual formulation. The primal Lagrangian function is described as follows:

$$\begin{aligned} L_P(\mathbf{w}, b, \Lambda) &= \frac{\|\mathbf{w}\|^2}{2} - \sum_{i=1}^{nb\_class} \lambda_i (y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1) \\ &= \frac{\|\mathbf{w}\|^2}{2} - \sum_{i=1}^{nb\_class} \lambda_i y_i \mathbf{w} \cdot \mathbf{x} - \sum_{i=1}^{nb\_class} \lambda_i y_i b + \sum_{i=1}^{nb\_class} \lambda_i \end{aligned} \tag{3.22}$$

where $\Lambda = (\lambda_1, \dots, \lambda_N)$ is the set of Lagrangian multipliers with $\lambda_i \geq 0$. The L_P must be minimized with respect to $\mathbf{w}$ and b and maximized with respect to λ_i. This is equivalent to maximize L_P subject to the constraints that the gradient of L_P with respect to $\mathbf{w}$ and b is zero and subject to the constraint $\lambda_i \geq 0$. Such constraints are named Karush-Kuhn-Tucker (KKT) conditions [68] and are as follows:

- Gradient Conditions :

$$\begin{aligned} \frac{\partial L_P(\mathbf{w}, b, \Lambda)}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{nb\_class} \lambda_i y_i \mathbf{x}_i = 0 \\ \frac{\partial L_P(\mathbf{w}, b, \Lambda)}{\partial b} &= \sum_{i=1}^{nb\_class} \lambda_i y_i = 0 \\ \frac{\partial L_P(\mathbf{w}, b, \Lambda)}{\partial \lambda_i} &= y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1 = 0 \end{aligned} \tag{3.23}$$

- Orthogonality Condition

$$\lambda_i g_i(\mathbf{x}) = \lambda_i \big(y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1 \big) = 0, \;\; i = 1, \dots, nb\_class \tag{3.24}$$

- Feasibility Condition

$$y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1 \geq 0, \;\; i = 1, \dots, nb\_class \tag{3.25}$$

- Non-negativity Condition

$$\lambda_i \geq 0, \;\; i = 1, \dots, nb\_class \tag{3.26}$$

Using the KKT conditions above, we can formulate the dual Lagrangian function L_D as follows:

$$\begin{aligned} &\text{maximize } L_D(\mathbf{w}, b, \Lambda) = \textstyle\sum_{i=1}^{nb\_class} \lambda_i - \frac{1}{2} \sum_{i=1}^{nb\_class} \sum_{j=1}^{nb\_class} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ &\text{subject to } C \geq \lambda_i \geq 0, i = 1, \dots, N \text{ and } \textstyle\sum_{i=1}^{nb\_class} \lambda_i y_i = 0 \end{aligned} \tag{3.27}$$

A point is an optimal point of Equation (3.27) if and only if the KKT conditions are satisfied and $\mathcal{Q}_{ij} = y_i y_j \mathbf{x}_i \mathbf{x}_j$ is positive semi-definite. The common method for solving this quadaratic optimization (QP) problem in (3.27) is the Sequential Minimal Optimization (SMO) algorithm proposed by Platt (1999) [95]. We will further describe this algorithm later in this section.

The points which have $\lambda_i > 0$ represent the support vectors. For those with $\lambda_i = 0$ are not important and can be removed. The vector $\mathbf{w}$ of the hyperplane is obtained by using Equation (3.23) such that $\mathbf{w} = \sum_{i=1}^{nb\_class} \lambda_i y_i \mathbf{x}_i$. The value b is computed by using the obtained $\mathbf{w}$ and Equation (3.24) such that $b = y_j - \sum_{i=1}^{nb\_class} \lambda_i y_i \mathbf{x}_i \cdot \mathbf{x}_j$

Once having the separation hyperplane, to predict the class value for any new point, we can use the sign following expression

$$\text{class of } (\mathbf{x_k}) = \text{sign } \Big(\sum_{i=1}^{nb\_class} \lambda_i y_i \mathbf{x}_i \cdot \mathbf{x}_k + b \Big) \tag{3.28}$$

We have introduced the SVMs classification algorithm for the linear problems. When the data are complex so that a linear classifier can not be used, then SVMs can be extended to handle the nonlinear surfaces using the nonlinear classifiers. In these cases, SVMs will map the complex data into a feature space (with the feature functions $\varphi(\cdot)$) of a higher dimension in order to use a linear classifier. Then the Equation (3.28) can be rewritten as follows

$$\text{class of } (\mathbf{x_k}) = \text{sign } \Big(\sum_{i=1}^{nb\_class} \lambda_i y_i \varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_k) + b \Big) \tag{3.29}$$

The inner product of the feature functions is computed using kernel method.

$$K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_k) \tag{3.30}$$

Some of the most used kernels are the polynomial kernel, Gaussian radial basis function, etc. In our option pricing problem we used the polynomial kernel because it is simple, efficient and most of all requires less time of computation. The polynomial kernel is as follows

$$K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^p \tag{3.31}$$

with p is the order of the polynomial. Much of computational time of SMO is spent for evaluating K.

As mentioned above, the common method for solving the QP problem in SVMs is the SMO algorithm proposed by Platt (1999) [95]. The SMO algorithm provides an advantage that it decomposes the overall QP problem into QP sub-problems. For each QP sub-problem, SMO chooses the smallest possible optimization problem which involves two Lagrange multipliers. Because solving two Lagrange multipliers can be done analytically.

The SMO algorithm consists of three components. The first one is an analytical method to solve for the two Lagrange multipliers. The second one is a heuristic step for choosing which multipliers to optimize. The last component is a method to compute the threshold value b. For finding the first multiplier, the algorithm will iterate over the entire dataset, finding the points which violate the KKT conditions. Whenever a point is found that it violates the KKT conditions then it is eligible for optimization. Once the first multiplier is found then SMO algorithm will go to find the second one to optimize. Once having two new multipliers, the SVM objective function will be updated with these two multipliers. After that SMO algorithm continues in finding two others multipliers that violate the KKT conditions to optimize them until all the Lagrange multipliers satisfy the KKT conditions. Beside, the computation of the threshold b is separated with solving the Lagrange multipliers. Each time when we have two new multipliers and update the objective function, b is also recomputed with this new objective function.

A Performance Comparison : The performance of classification algorithms of course depends on many different parameters such as the dataset size, number of dimensions of data, the underlying learner function of each classification algorithm, etc. We aim to compare the performance of three algorithms mentioned above in term of classification accuracy and computational time. Such comparisons have been studied before. For example, Meyer and al. (2003) in [87] compared the SVMs algorithm to 16 classification methods and 9 regression methods on real and standard datasets in term of accuracy. However, such methods do not include boosting algorithm using decision tree and as conclusion, the authors stated that SVMs featured good performance but was not top ranked on all datasets. In Tan and Gilbert (2003) [112], the authors focused on comparing 3 classification algorithms using a single decision tree, bagging of decision tree (bootstrap aggregating) and boosting of decision tree (Adaptive Boosting: AdaBoost) for the cancer classification problem. Of course the performance of bagging and boosting are better than a single tree. Later, Sordo and Zeng (2005) [106] investigated the link between the classification accuracy and the dataset size of three classification algorithms including SVMs and a single decision tree. While increasing the dataset size up to 8000 points, the SVMs more and more outperformed the single decision tree in term of accuracy.

In our option pricing problem, the computational time is an overall very important factor hence we want to figure out the dependency between it and the classification accuracy. We perform the CMC algorithm for our high dimensional American option pricing problem with all of three algorithms including AdaBoost, Gradient Boost and SVMs. In comparing the estimated results with other numerical reference results, we can figure out the accuracy of the option prices. Hence, we can compare indirectly the performance of each classification algorithm regarding the computational time and accuracy trade-offs, see further in Section 3.4.3 for more details.

3.4.2 Parallel Monte Carlo Phases Approach

Recall Algorithm 6 that illustrates the serial approach based on CMC Algorithm. Notice that at a given date t_m, knowing S_{t_m} it is possible to simulate $S_{t_{m-1}}$ using the Brownian bridge. In case of d dimensional option pricing, each trajectory implies of d sub-trajectories, one for each asset. Since all these d are always simulated together hence, we will not consider the factor d into the complexity of the algorithm. For simplicity we assume a master-worker programming model for the parallel implementation: the master is responsible for allocating independent tasks to workers and for collecting the results.

Regarding Algorithm 6, the [**phase 1**] contains two sub-steps of computations which are [**step 1**] and [**step 2**]. The complexity of [**phase 1**] can be described as follow

$$\mathcal{O}\left(\sum_m m \times nb\_class \times \sum_m (N-m) \times nb\_cont \times \mathcal{O}(\mathbf{predict}) + \sum_m \mathcal{O}([\mathbf{step\ \ 2}])\right), m = N-1, \dots, 1. \tag{3.32}$$

where $\mathcal{O}(\mathbf{predict})$ is the required time to predict the label value of one given data point. Notice that $\mathcal{O}(\mathbf{predict})$ is not always the same but strongly depends on the given classification algorithm. Based on our experiment in the next section, we observe different values of $\mathcal{O}(\mathbf{predict})$ for Gradient Logit Boost, AdaBoost and SVMs respectively. This observation is very important because it directly influence to the overall computational time of the CMC algorithm. We will further return to this comment in the performance analysis section.

At the end of [**step 1**], the master will collect all *nb_class* points from workers and goes to the [**step 2**]. The computation in [**step 2**] is serially performed by the master. The reason is that since the portion of computational time of the classification step is very small comparing with the whole [**phase 1**] and the entire algorithm, see Figure 3.10, thus parallelizing such [**step 2**] does not gain much acceleration for the entire algorithm. Moreover as we mentioned earlier in this chapter, the [**step 2**] is often hard to be parallelized.

Hence, we have two parallelization strategies for this [**phase 1**] excluding the [**step 2**]. The first one is to distribute the *nb_class* points and keeps the *nb_cont* trajectories run sequentially. The second one could be the parallelization of *nb_cont* trajectories.

In the context of this thesis, we only implemented the first strategy. In this case, *nb_class* points are divided into *nb_tasks* tasks then are distributed to a number of workers. Thus each worker has $\dfrac{nb\_class}{nb\_tasks}$ points. Now the complexity of [**phase 1**] will be :

$$\mathcal{O}\left(\sum_m m\times\frac{nb\_class}{nb\_tasks}\times\sum_m(N-m)\times nb\_cont\times\mathcal{O}(\mathbf{predict})+\sum_m\mathcal{O}([\mathbf{step}\ \ \mathbf{2}])\right), m = N-1,\dots,1. \tag{3.33}$$

The factor $\dfrac{nb\_class}{nb\_tasks}$ should be as small as possible. Ideally, each worker should be only assigned 1 point to do which means $\dfrac{nb\_class}{nb\_tasks} = 1$.

For the last phase of computing the final results [**phase 2**] which is embarrassingly parallel, naturally the number *nbMC* of simulations are divided into *nb_tasks* tasks (this *nb_tasks* could differ from the number of tasks in [**phase 1**]) and then are distributed across the workers. In case of homogeneous system, we set the number of tasks *nb_tasks* be equal with the number of workers. Hence, each worker has $\dfrac{nbMC}{nb\_tasks}$ simulations to do. Finally we gain a linear speedup of $\dfrac{nbMC}{nb\_tasks}$. Otherwise in case of heterogeneous system, each worker should be assigned a small number of simulations in order to profit the "aggressive" dynamic task distribution mechanism discussed in Chapter 2. The entire parallel approach for the CMC algorithm is described in Algorithm 9 below.

In the next section, we are going to present the numerical experiments and the performance results obtained with the parallel CMC algorithm in a grid environment using PicsouGrid.

3.4.3 Numerical Experiments

This section presents numerical experiments for the parallel CMC algorithm. Each experiment will be performed with three classification algorithm including AdaBoost, Gradient Boost and SVMs. As we mentioned earlier we attempted to price the American options on d underlying assets (with d up to 40) and to our knowledge no such results have been published earlier. Let us recall the Black Scholes equation (BSE) (1.17) for modelling the asset prices as we described in Chapter 1

$$dS_t^i = S_t^i r dt + S_t^i \sum_{k=1}^{d} a_{ik} dW_t^k, i = 1,\dots,d$$

In our experimentations, we consider the S^i are a family of independent stochastic processes. This means that $a_{ik} = 0$, for $i \neq j$, moreover for a given $\sigma > 0$ we set $a_{ii} = \sigma$, $\forall i = 1,\dots d$.

Algorithm 9 Parallel Classification and Monte Carlo Algorithm

```
Require: $S_0^i$, $d$, $r$, $\delta_i$, $\sigma_i$, $T$, $N$, number of tasks $nb\_tasks$.
Require: number of classification points $nb\_class$,
Require: number of trajectories to estimate the continuation value $nb\_cont$,
Require: number of simulations to estimate the final option price $nbMC$.
 1: [phase 1] : // Characterize the optimal exercise boundary at every opportunity date.
 2: for $m = N-1$ to 1 do
 3:   In parallel do : generate $\frac{nb\_class}{nb\_tasks}$ points, $\{S_{t_m}^{i,(s)} : i = 1, \ldots, d; s = 1, \ldots, \frac{nb\_class}{nb\_tasks}\}$.
 4:   [step 1] : // Compute the continuation value and the exercise value to obtain the
      decision of exercise or not.
 5:   for $s = 1$ to $\frac{nb\_class}{nb\_tasks}$ in parallel do
 6:     [step 1'] : // is serially done on each worker.
 7:     Compute $C^{(s)}(S_{t_m}, t_m) = \mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}}, t_{m+1})|S_{t_m}\big]$ using $nb\_cont$ trajec-
        tories and also compute $\Psi^{(s)}(S_{t_m}, t_m)$.
 8:     if $C^{(s)}(S_{t_m}, t_m) \leq \Psi^{(s)}(S_{t_m}, t_m)$ then
 9:       sign = 1
10:     else
11:       sign = -1
12:     end if
13:   end for
14:   [step 2] : Merging on the master the computed classification points from each worker
      $\Big\{\big(S_{t_m}, \mathrm{sign}\big)^{(s)} : s = 1, \ldots, nb\_class\Big\}$ for characterization of the exercise boundary
      $F(t_m)$.
15: end for
16: [phase 2] : Using the characterization of the exercise boundary $F(t_m) : m = 1, \ldots, N-$
    $1$, each worker simulates $\frac{nbMC}{nb\_tasks}$ simulations to compute the partial results. The
    master will merge all of partial results to estimate the final option price.
17: return the final option price.
```

3.4.3.1 Testcases Description

- **One Dimensional American Put :** Valuing this one dimensional option helps us to compare the results obtained by parallel CMC algorithm with the true prices which are computed by common low dimensional methods for option pricing such as binominal lattice method [37, 62]. The option parameters are as follows:

$$\begin{aligned} &S_0 = 36, K = 40, r = 0.06, \delta = 0.0, \sigma = 0.4, \\ &\text{maturity date } T = 1 \text{ year} \end{aligned} \tag{3.34}$$

- **Geometric Average American Call Option :** Consider a call American option on the geometric average of d assets. The asset prices follow the BSE (1.17). The call payoff at time t is defined as $\Phi(S_t, t) = (\sqrt[d]{\Pi_i S_t^i} - K)^+$. The option paramteters are given below :

$$\begin{aligned} &S_0^i = 90, K = 100, r = 0.03, \delta = 0.05, \sigma = 0.4, \\ &\text{maturity date } T = 1 \text{ year} \end{aligned} \tag{3.35}$$

  Such geometric average option of d assets can be reduced to an option that depends only on one underlying factor. Therefore in this case the resulting one-dimensional option can be accurately priced. Consider the option on 7 underlying asset ($d = 7$), the option with new reduced asset has the following parameters:

$$\begin{aligned} &\widehat{S}_0 = 90, \widehat{r} = 0.03, \widehat{\delta} = \delta - \tfrac{\sigma^2}{2\times d} + \tfrac{\sigma^2}{2} = 0.1186, \\ &\widehat{\sigma} = \tfrac{\sigma}{\sqrt{d}} = 0.1512, \rho = 0, T = 1 \text{ year} \end{aligned} \tag{3.36}$$

- **Maximum American Call Option :** Consider a call American option on the maximum of d assets. The asset prices follow the BSE (1.17). The call payoff at time t is defined as $\Phi(S_t, t) = (\max_{i=1,..,d}(S_t^i) - K)^+$. The option paramteters are as follows :

$$\begin{aligned} &S_0^i = 90, K = 100, r = 0.05, \delta = 0.1, \sigma = 0.2, \\ &\text{maturity date } T = 3 \text{ years} \end{aligned} \tag{3.37}$$

- **Minimum American Put Option :** Consider a put American option on the minimum of d assets. The asset prices follow the BSE (1.17). The call payoff at time t is defined as $\Phi(S_t, t) = (\min_{i=1,..,d}(K - S_t^i))^+$ The option parameters are as follows :

$$\begin{aligned} &S_0^i = 100, K = 100, r = 0.06, \delta = 0, \sigma = 0.6, \\ &\text{and maturity date } T = 0.5 \text{ years} \end{aligned} \tag{3.38}$$

3.4.3.2 Convergence Results Analysis

In [94], Picazo originally performed the option pricing testcases with $nb\_class = 5000$ and $nb\_cont = 100$ ($nb\_cont = 50$ in the case of one-dimensional American put option pricing). All the testcases were computed with a small number of opportunity dates (smaller than 13) except the one-dimensional American put option which was computed with 50 opportunity dates. Moreover, the reference results which were used to compare with the estimated option prices obtained by CMC algorithm in [94], were also reported with the same small number of opportunity dates.

It is clear that the convergence of option prices computed by CMC algorithm may depend on several parameters such as the number of classification points nb_class, the number of trajectories for continuation values computation nb_cont, the chosen classification algorithm and the number of opportunity dates N. Therefore, in our experiments, we perform the option pricing with some modifications in parameters configuration in comparison with the original ones of Picazo [94].

Firstly, we observed that $nb\_cont = 100$ is not enough and we increased it to 10000. The impact of this change is figured out in the case of single American put pricing. Secondly, it is clear that the Bermudan option price will converge to the American one when the number of opportunity dates N is big enough. Hence, in order to observe this convergence, we computed our option prices with up to 100s opportunity dates. Next, each testcase is performed with three classification algorithms. Finally, in our figures, the option prices which are produced by the CMC algorithm are presented within a confidence interval. We use $nbMC = 2 \times 10^6$ of Monte Carlo simulations for the final pricing phase for every testcase (except in the case of one dimensional American put, we use $nbMC = 10^6$).

The parameters of CMC algorithm were chosen as follows:

$$\begin{aligned}
&nb\_class = 5000,\ nb\_cont = 10000,\\
&\text{Boosting iterations K} = 150,\\
&\text{Number of Monte Carlo simulations } nbMC = 2 \times 10^6,\\
&\text{Decision tree classifier for AdaBoost and Gradient Logit Boost.}\\
&\text{Linear and Polynomial kernel for SVMs.}
\end{aligned} \tag{3.39}$$

- **One Dimensional American Put :** Consider the put option which was described in (3.34). In [37, 62], the price of such option is reported as 7.109 using binominal tree method with $N = 250$ opportunities ($m = 1, \ldots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 1$ year). Since the number of opportunity dates is big enough, we consider this price as the American price and present it by the black continuous line in Figure 3.3. In order to have more reference results, we also use the LSM algorithm of Longstaff and Schwartz which is implemented in Premia[2], to compute this option price. As expected, the obtained prices by LSM algorithm (using 10^7 simulations) converge to the American price while increasing N up to 200, see the green continuous curve in Figure 3.3. In fact, the option prices produced by LSM algorithm have to be presented within a confidence interval (CI) however, because Premia does not provide such CI values thus the green curve only presents the prices.

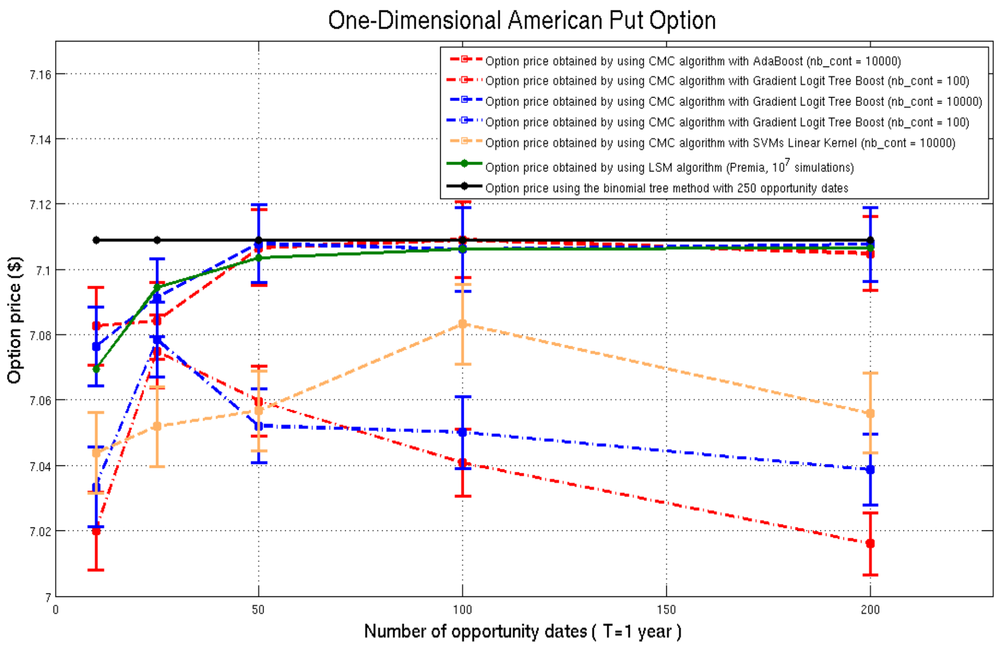

Figure 3.3: One Dimensional American Put option.

Figure 3.3 shows the prices produced by CMC algorithm with three classification algorithms. For each classification algorithm, the option prices are computed with different numbers of opportunity date, starting from 10 up to 200. As we can observe, the value of N has a strong impact in computing the option price. Similarly as in case of LSM algorithm, when increasing N the option prices produced by CMC algorithm will converge to the American price. However, only the experiments with AdaBoost and Gradient Tree Logit Boost present a good convergence (the red and blue dash curves). Meanwhile, in the case of SVMs, we observe a poor result (the orange dash curve). The reason is that the kernels used within SVMs are designed to support the

[2]www-rocq.inria.fr/mathfi/Premia/index.html

high dimensional classification problems. Even we have chosen the linear kernel for our experiments, it is still not a suitable kernel for this one dimensional classification problem.

Moreover, in term of the impact of *nb_cont*, we can see that the prices obtained with *nb_cont* $= 10000$ (the dash curves) is much better than the case of *nb_cont* $= 100$ (the dash dot curves) for all of three classification algorithms. Even with $N = 200$, the experiments with *nb_cont* $= 100$ still resulted undesired option prices. Based on such observation thus from now, the rest of our experiments with other option types will be performed with *nb_cont* $= 10000$.

- **Geometric Average American Call Option :** The generic geometric average (GMA) option on d assets was described in Section 3.4.3. In our experiment, we choose $d = 7$. As we mentioned earlier, such option pricing testcase is a useful example for validating any high dimensional American option pricing algorithm. Because such geometric average option of d assets can be reduced to a "reduced" option that depends only on one underlying factor and can be computed by many common methods in American option pricing (e.g lattice based methods, Monte Carlo based methods). Hence we can easy compare the estimated high dimensional option price with the "reduced" one. Using the dimension reduction technique mentioned in Chapter 1, the "reduced" underlying asset from basket of 7 assets was described in (3.36).

  Such "reduced" option was studied in Broadie and Glasserman (2004) [26]. The authors reported a price of 0.761 using the binomial tree method with $N = 10$ opportunity dates ($m = 1, \dots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 1$ year). This price is presented by the black continuous line in Figure 3.4. In order to obtain more reference results with different number of opportunity dates, we perform such "reduced" option pricing using two others algorithms implemented in Premia. The first one is the Euler binomial tree method and the second one is the LSM algorithm (using 10^7 simulations). We have chosen the LSM algorithm as a reference for this case because it is a "reduced" one-dimensional American option pricing. The number of opportunities for Euler binomial tree method is 50 and 100 while for LSM algorithm $N = 10, 25, 50, 100$. In Figure 3.4, the two brown continuous lines present the prices obtained by Euler method with 50 and 100 opportunities respectively. As before, the green continuous curve presents the prices obtained by LSM algorithm. We can observe that the green continuous curve approaches the black lines at $N = 10$ and approaches the two brown lines at $N = 50$ and $N = 100$. That means the reported price from Broadie and Glasserman (2004) [26] is only a good reference as a Bermudan option with 10

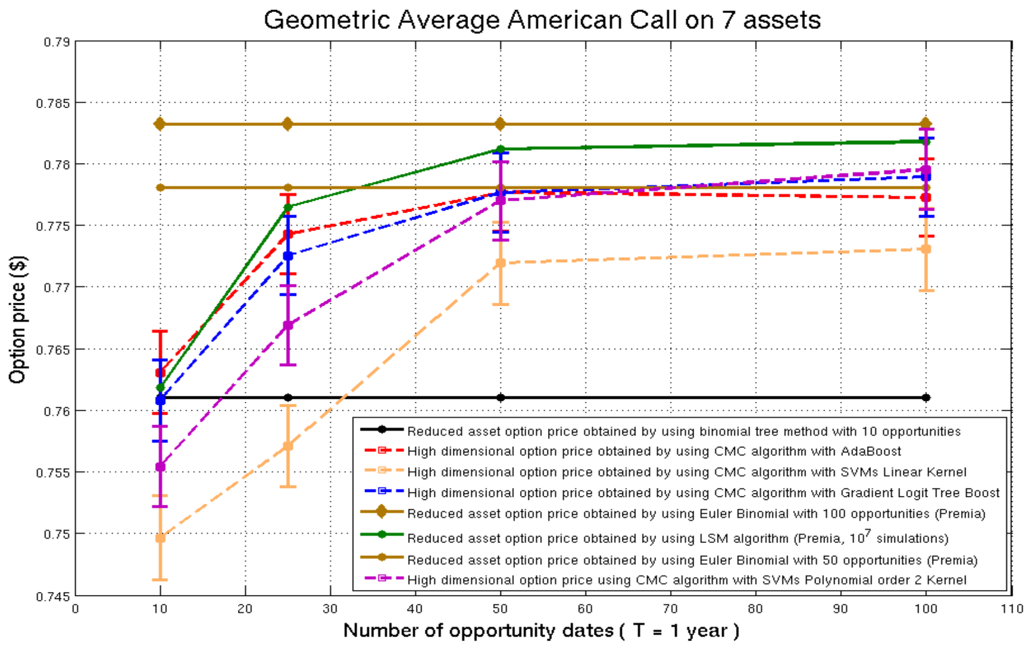

Figure 3.4: Geometric Average American Call option on 7 assets.

opportunity dates.

As we can observe in Figure 3.4, it is similar as in the case of the one dimensional American put option pricing above (e.g. the convergence of option price has a significant dependency on the increasing of N). The red, blue, violet and orange dash curves present the option prices using CMC algorithm with AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel and SVMs linear kernel respectively. The red, blue and violet dash curves converge to the green continuous one and the two brown continuous lines with respect to the number of N. Meanwhile, the orange dash curve converges less than the three other dash curves. That means the AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel perform the CMC algorithm better than the SVMs linear kernel in this option pricing. Among the three good dash curves, it seems that the SVMs polynomial order 2-kernel overcomes AdaBoost and Gradient Logit Tree Boost at $N = 100$. However, it is difficult to state which algorithm is the best trade-off regarding the confidence interval for each price versus spent computational effort. Hence in order to do that, we need to figure out the performance in term of computational time for each algorithm which will be discussed further in this section.

- **Maximum American Call Option :** Note that we do not have an exact price for this kind of option. This kind of option on 5 assets has been studied in several works [26, 63, 5]. First, Broadie and Glasserman (2004) [26] applied the stochastic mesh method to compute such option price with $N = 9$ ($m = 1, \ldots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 3$ years) and reported the 95% CI for this option as [16.602, 16.710] (presented by the two black continuous lines in Figure 3.5). As before, in order to obtain more reference results with different numbers of opportunity dates, we perform this high dimensional option pricing using two others algorithms implemented in Premia: the LSM algorithm (using 10^7 simulations) and the Tsitsiklis-VanRoy algorithm [119] (using 5×10^4 simulations) with $N = 9, 30,$ and 75, respectively. The prices obtained by these two algorithms are presented by the green and brown continuous curve in Figure 3.5, respectively.

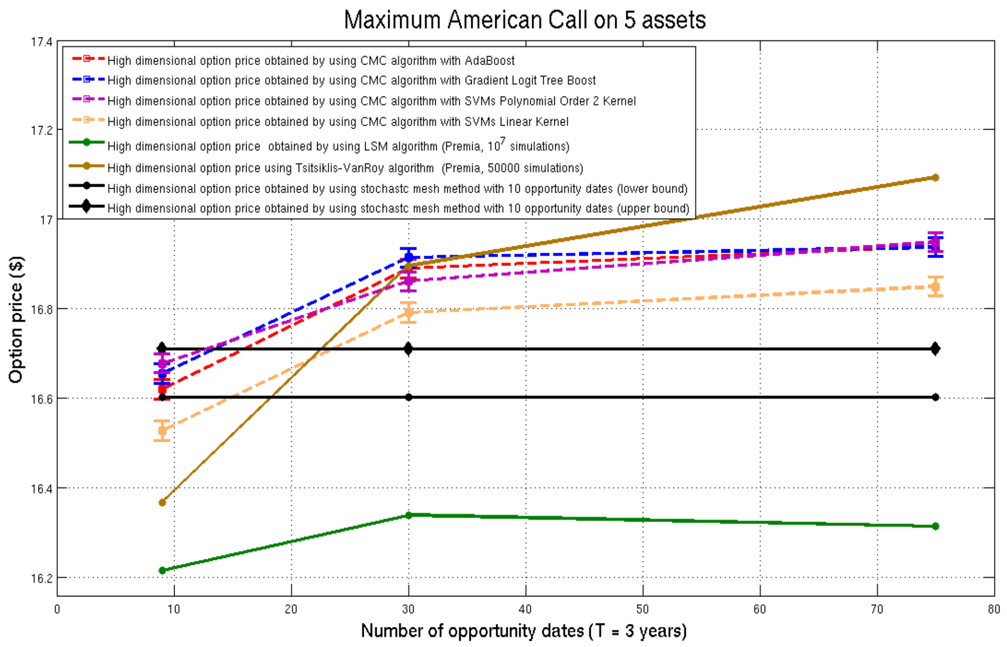

Figure 3.5: Maximum American Call option on 5 assets.

As we can observe in Figure 3.5, both Tsitsiklis-VanRoy and LSM algorithm produced an undesired result in comparison with the bounds provided by Broadie and Glasserman at $N = 9$. Moreover, the LSM algorithm then produced decreasing prices while increasing N.

In term of CMC algorithm, the red, blue, violet and orange dash curves present the option prices obtained with AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel and SVMs linear kernel respectively. At $N = 9$, all of the prices obtained by CMC algorithm (except the one with SVM linear kernel) are within the lower and upper bound provided by Broadie and Glasserman (2004) [26]. When increasing N, the prices obtained with AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel grow up like the results of Tsitsiklis-VanRoy algorithm. As we can observe, these prices seem to converge to one point at $N = 75$.

As in the testcase of geometric average option pricing, it is not easy to conclude a best choice among 3 classification algorithms and we need to analyze the computational time for each algorithm.

- **Minimum American Put Option :** As in the case of maximum option, we do not have an exact price for this kind of option. However, the minimum option on 10 assets was studied in Roger (2002) [99] and the price is reported as 48.33 using duality method with $N = 50$ ($m = 1, \dots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 0.5$ year). We performed this high dimensional option pricing using two others algorithms implemented in Premia: the LSM algorithm (using only 10^6 simulations due to the memory limitation of Premia because now we have $d = 10$) and the Tsitsiklis-VanRoy algorithm (using 50000 simulations) with $N = 12, 25, 50,$ and 100 (presented by the green and brown continuous curves in Figure 3.6, respectively).

  We have the same observation as in case of maximum option that LSM algorithm produced an undesired result in comparison with the bounds provided by Roger at $N = 50$. Moreover, the LSM algorithm then produced decreasing prices while increasing N. Meanwhile the prices obtained by Tsitsiklis-VanRoy seem to approach with the lower bound of Roger at any value of N.

  In term of CMC algorithm, the red, blue, violet and orange dash curves present the option prices with AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel and SVMs linear kernel respectively. At $N = 12$, all of the prices obtained by CMC algorithm are quite out of the lower and upper bound provided by Roger. However, once N increases, the prices obtained with AdaBoost, Gradient Logit Tree Boost, SVMs polynomial order 2-kernel grow up and are within the bounds of Roger. Finally, as we can observe, these prices obtained by CMC algorithm converge to one point at $N = 100$.

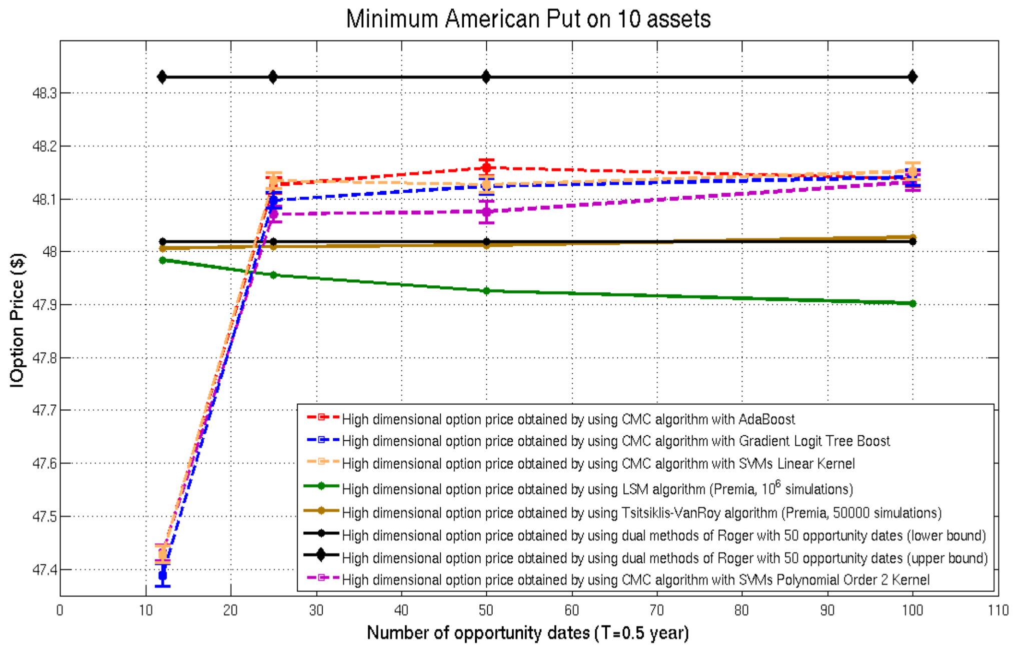

Figure 3.6: Minimum American Put option on 10 assets.

3.4.3.3 Computational Time and Speedup Analysis

Computational Time : Since, it is difficult to state which algorithm is the best regarding numerical convergence analysis above, we are interested in investigating overall computational time in order to try to establish acceptable trade-off time/accuracy for each classification algorithm.

For the case of one dimensional American put option, obviously the CMC algorithm is not a good choice due to the existence of other low dimensional analytical methods (e.g. the PDE, tree methods). Therefore we do not investigate the computational time of the CMC algorithm for this one dimensional case.

For the other high dimensional testcases, first let us consider the Geometric Average American Call option pricing on 7 assets. The Figure 3.7 shows the computational time for each classification algorithm with respect to the number of opportunity dates. Each experiment was performed using 64 processors (each one is a Intel(R) Xeon(R) 2.00 Ghz CPU, 2G RAM). The computational times are reported in minutes. As we can observe, the computational time increases with increasing of N.

The Figure 3.8 shows the computational time for pricing the Maximum American Call

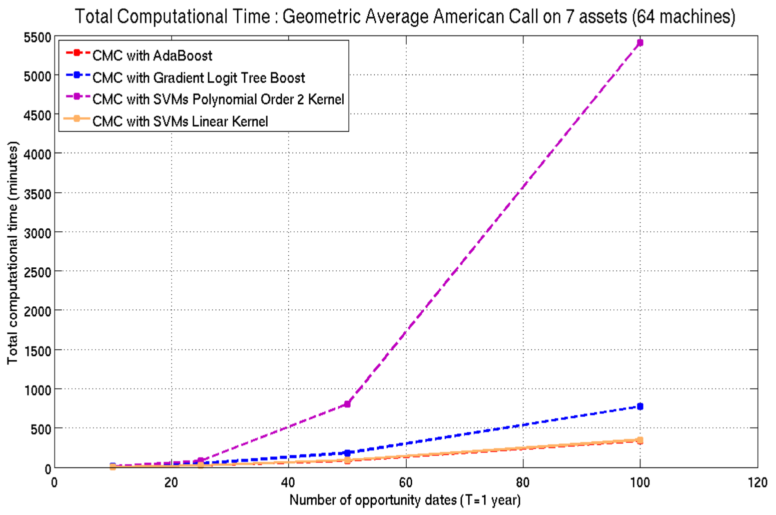

Figure 3.7: Total computational time for pricing the Geometric Average American Call option on 7 assets using 64 processors.

option on 5 assets. The Figure 3.9 shows the computational time for pricing the Minimum American Put option on 10 assets.

Speedup : Because our parallelization strategy does not depend on the choice of option type thus we consider the Geometric Average testcase to analyze the speedup. First, we figure out the distribution of computational time for each phase of the CMC algorithm, see Figure 3.10. We perform this experiment on a single machine (Intel 2 Duo 2.20 Ghz CPU, 3.5G RAM). The total computational time is 220, 259, 1132 minutes for AdaBoost, Gradient Logit Tree Boost and SVMs with polynomial order 2 kernel, respectively. The SVMs with linear kernel is not considered here due to its poor numerical convergence in almost testcases.

On the experiments plotted in Figure 3.10 we can observe that it is [**phase 1**] (the red part) which consumes most of the time for overall computation (almost 99% of total time for all of three algorithms). The time for performing the [**step 2**] only takes a very small portion of the total time ($\ll 1\%$). The percentage of computational time of [**phase 2**] approximates 1% of total time (the yellow part). However, as we can see in the original complexity in (3.32) and after parallelizing in (3.33) when m decreases in each backward induction steps, $(N-m)$ increases linearly. Then the amount of computational time of [**step 1'**] $\left(\sum_m (N-m) \times nb\_cont \times \mathcal{O}(\mathbf{predict})\right)$ increases linearly and consumes a significant

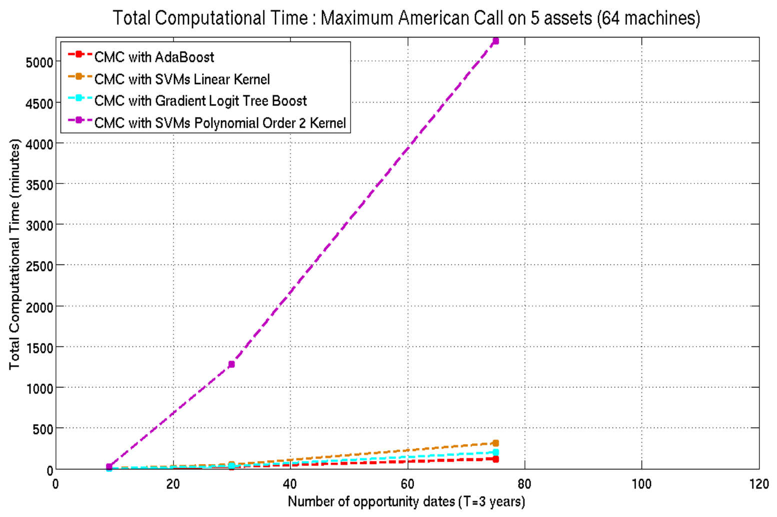

Figure 3.8: Total computational time for pricing the Maximum American Call option on 5 assets using 64 processors.

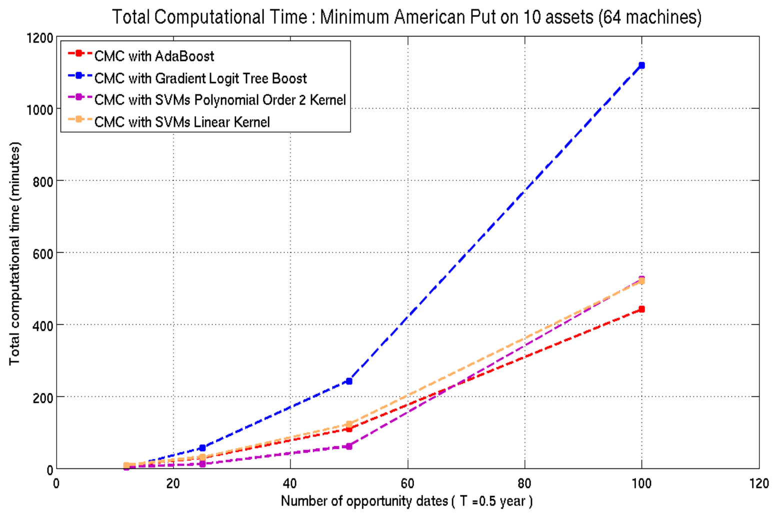

Figure 3.9: Total computational time for pricing the Minimum American Put option on 10 assets using 64 processors.

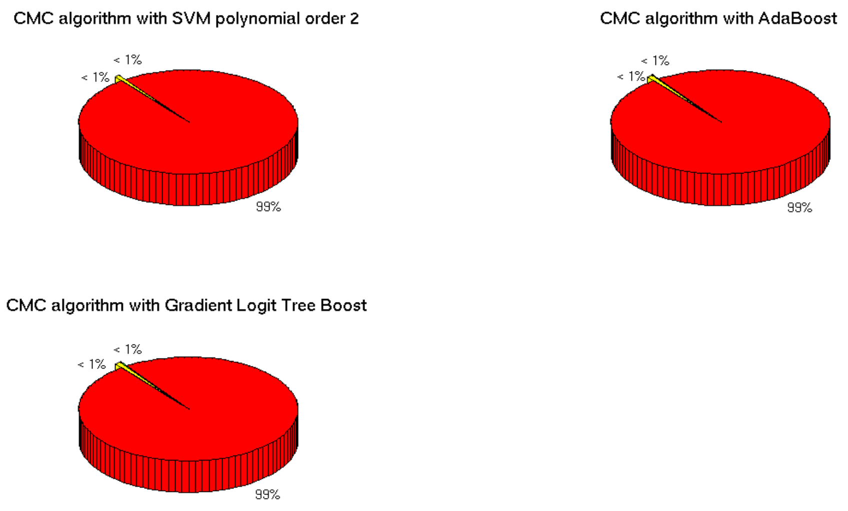

Figure 3.10: Distribution of computational time for each phase of the CMC algorithm in case of pricing Geometric Average American Call option on 7 assets with $nb\_cont = 10000$.

amount of the total computation time. Unfortunately, such computation is serially run on each worker. We assume that parallelizing this [**step 1'**] would accelerate very much the total computational time of the entire algorithm. However, since we already distributed the *nb_class* points to our workers it is not possible to distribute again the *nb_cont* trajectories to such workers because they are already all busy. It is possible only in case where the number of workers is very much bigger than *nb_class*. As mentioned above, we can use the strategy which parallelizes [**step 1'**] instead of distributing the *nb_class* points. But this problem is not considered within this thesis and this can be considered as future work.

Figure 3.11 presents the speedup which was achieved by the parallel approach for the CMC algorithm. Such approach is based on the distribution of *nb_class* points across processors (aka. workers).

We can observe that the parallel approach achieves almost linear speedup with 64 processors. As we have shown in the figures, different parts of the algorithm scale differently. In our particular approach, only a part of [**phase 1**] and [**phase 2**] are parallelized and they scale almost linearly. There is still a limit in our parallel approach which is the sequential part of [**step 2**] hence it is constant despite of the number of processors. However, this only takes a small amount of time of total computational time, see Figure 3.10. According to this, we can see that this factor does not make an effect on the overall speedup.

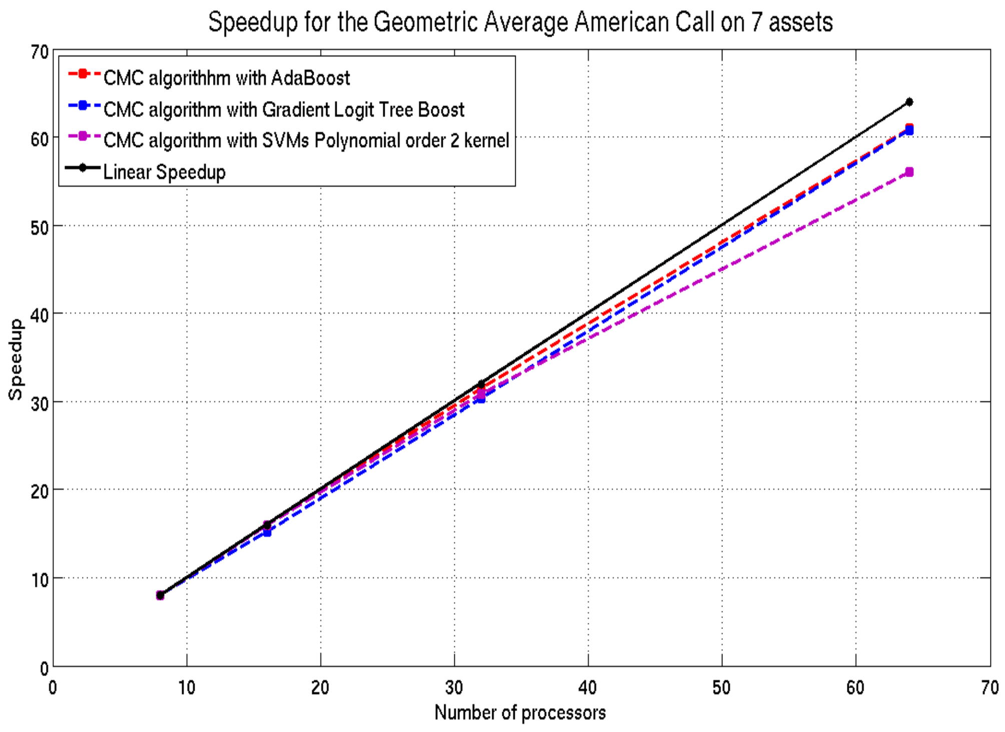

Figure 3.11: Speedup for pricing the Geometric Average American Call option on 7 assets.

3.4.3.4 American Option Pricing on 40 Assets: Numerical Results

Pricing the large dimensional (e.g. $d > 10$) options particularly American ones, is still a challenge in term of practical computational time. Even the pricing library Premia, which we used to obtain the reference results in Section 3.4.3, is limited itself with $d \leq 10$. Thanks to the use of PicsouGrid, we are now able to run such large dimensional American options pricing up to 40 underlying assets (e.g. based on the French CAC40 stock index). We, in order, performed the geometric average call/put, maximum call and minimum put American options on 40 underlying assets using the CMC algorithm with AdaBoost. We choose AdaBoost for such 40-dimension pricing problems because the experiments, presented in Section 3.4.3.2 and 3.4.3.3, using AdaBoost offered a good trade-off between numerical convergence and computational time. The parameters of CMC algorithm were chosen as follows:

$$\begin{aligned}&nb\_class = 5000,\ nb\_cont = 10000,\\ &\text{Boosting iterations K} = 150,\\ &\text{Number of Monte Carlo simulations } nbMC = 2\times 10^6,\\ &\text{Decision tree classifier for AdaBoost.}\end{aligned} \tag{3.40}$$

- **Geometric Average American Call/Put Option :** The option paramteters are as follows :

  $$S_0^i = 100, i = 1,\dots,40;\ K = 100,\ r = 0.03,\ \delta = 0.05,\ \sigma = 0.4,\ T = 1 \text{ year} \tag{3.41}$$

  Such call and put options are computed with $N = 50$ opportunity dates ($m = 1,\dots,N$, where $t_m = \frac{m\times T}{N}$ and $T = 1$ year) which means the option holder can exercise his option contract every one-week until the maturity date. As before, in this geometric average case, we also provide the option based on the "reduced" asset (with $d = 40$), see below:

  $$\begin{aligned}&\widehat{S}_0 = 100,\ K = 100,\ \widehat{r} = 0.03, \widehat{\delta} = 0.128,\\ &\widehat{\sigma} = 0.0632, \rho = 0, T = 1 \text{ year}\end{aligned} \tag{3.42}$$

  Table 3.3 presents the option prices computed by CMC algorithm and the reference option prices based on the one dimensional "reduced" asset obtained by LSM and Euler binomial algorithms in Premia. The overall computational time for the CMC algorithm is 8.82 hours using 64 processors

- **Maximum American Call Option :** In this case ($d = 40$), we do not have an exact price for this kind of option. No equivalent experiments have been published in the

Option	CMC	LSM (10^7 simulations)	Euler
Call	0.70571 (± 0.0013)	0.70392	0.69469
Put	9.21329 (± 0.00714)	9.21833	9.21846

Table 3.3: The geometric average American call/put options pricing on 40 assets. The column CMC presents the high dimensional option prices obtained by the CMC algorithm. The LSM and Euler columns report the option prices based on the one dimensional "reduced" asset, computed by the LSM and Euler binomial algorithm respectively. All prices are computed with 50 opportunity dates.

literature. Even the current version of Premia (the public version 9) does not support such high dimensional options. The option paramteters are as follows :

$$S_0^i = 100, i = 1, \ldots, 40;\ K = 100,\ r = 0.05,\ \delta = 0.1,\ \sigma = 0.2,\ T = 3 \text{ years} \quad (3.43)$$

We compute this call option with $N = 30$ opportunity dates ($m = 1, \ldots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 3$ years). The price is reported with 95% CI as: 65.86631±0.02965. The overall computational time is 3.73 hours using 64 processors.

- **Minimum American Put Option :** Similarly as in case of the maximum option ($d = 40$), no exact price, neither equivalent experiments have been published in the literature. The option paramteters are as follows :

$$S_0^i = 100, i = 1, \ldots, 40;\ K = 100,\ r = 0.06,\ \delta = 0.0,\ \sigma = 0.6,\ T = 0.5 \text{ year} \quad (3.44)$$

We compute this call option with $N = 50$ opportunity dates ($m = 1, \ldots, N$, where $t_m = \frac{m \times T}{N}$ and $T = 0.5$ years). The price is reported with 95% CI as: 59.83627 ± 0.00926. The overall computational time is 11.5 hours using 64 processors.

3.5 Conclusion

In this chapter, we particularly focused on the Classification-Monte Carlo (CMC) algorithm for pricing high-dimensional American options. CMC algorithm belongs to the regression based methods for option pricing, however instead of performing a regression as the Least Square Monte Carlo algorithm of Longstaff and Schwartz, the CMC algorithm aims to estimate the characterization of the exercise boundary of the American option using a generic classification technique. After that, using such characterization of the exercise boundary, the CMC algorithm simulates straightforward Monte Carlo simulations in order to estimate the option price.

It is well-known that the Monte Carlo based methods in option pricing is very useful in case of high dimensional problems, however they are not practical in term of computational time. Hence our contribution was to provide a parallel approach for the CMC algorithm. We evaluated scalability of the proposed parallel approach in a computational grid environment. We also experimented with some real size world options with large number of underlying assets (e.g. 40 underlying assets). In terms of validation of our implementation, we performed the parallel CMC algorithm for pricing different option types such as geometric average, maximum or minimum basket options. To our knowledge, all the high dimensional American option pricing examples in the academic literature are examinated with a small number of exercise dates. The reason might be due to the large amount of required computational time however that leads to the less accurate results as we could confirm by the experiments. In our experimentations, in order to obtain high accuracy results, we performed our American option pricing examples with a high number of exercise dates.

We also analyzed the performance and the accuracy of the CMC algorithm with several classification algorithms such as AdaBoost, Gradient Boost and Support Vector Machines. All of the methods have performance–accuracy tradeoffs. These performance results are still far from a longer term goal which aims to price these options within a critical time (i.e. within hand of minutes).

In term of perspectives, firstly we could improve the parallelism strategy for the [**phase 1**] where we aim to parallelize the continuation values computation [**step 1'**] instead of distributing the *nb_class* points. Then secondly, we want to implement a parallel version for the [**step 2**] which performs the classifier model. In fact, there exists several parallel implementations for Support Vector Machines [54, 28], for AdaBoost [50] and for general boosting algorithms [72, 78]. However, these parallel approaches focus on using the shared memory paradigm which should thus require to be implemented in PicsouGrid. Since ProActive itself does not include any built-in shared data mechanism, hence such an extension should better be first addressed at the ProActive programming model level.

Chapter 4

FINANCIAL BENCHMARK SUITE FOR GRID COMPUTING

This chapter aims to introduce the definition and the use of a financial benchmark suite used for performance evaluation and analysis of grid computing middleware. Such benchmark suite was successfully used in the 2008 SuperQuant Monte Carlo challenge - the Fifth Grid Plugtest and Contest using the ProActive Monte Carlo API (part of the ProActive Parallel Suite). Such research work was reported in the INRIA technical report [41] and later was also presented in 4th *International Conference on High Performance Scientific Computing in 2009 (full paper is still under review for inclusion in the final proceedings)*

4.1 Introduction

As we discussed in the first two chapters of this thesis, financial engineering has become a critical discipline and have gained strategic reputation for its own. Financial mathematicians (aka. financial quant) keep coming up with novel and complex financial products and numerical computing techniques which often increase volume of data or computational time while posing critical time constraints for transactional processing. Interestingly, that raised interest towards the use of grid computing in computational finance. In practice, financial institutions are more and more contemplating using grid computing to perform more time critical computations for competitive advantage even if today they mostly rely on in-house large size HPC clusters. With this unprecedented computational capacity, running overnight batch processes for risk management or middle-office functions to re-evaluate whole product of portfolios is now affordable. However, managing such powerful but heterogeneous computing environment has never been an easy problem and obviously, that requires tools and middleware to do that (e.g. PicsouGrid presented in Chapter 2 is one such example).

In that sense, grid middleware is what makes grid computing work and easier to work with. It provides abstractions for core functionalities like authentication across large number of resources, authorization, resource matchmaking, data transfer, monitoring and fault–tolerance mechanisms in order to account for failure of resources etc. Any robust financial service operation cannot be achieved without paying a great attention to such issues. Grid middleware research and development had its beginning in the Condor Project[1] and the Globus Alliance[2]. Recently, we have seen an upsurge of academic and commercial middleware solutions such as gLite[3], ProActive/GCM Parallel Suite[4], Alchemi .NET grid computing framework[5], Unicore[6] and KAAPI/TakTuk[7]... Now the question is which middleware to choose for gridifying financial applications? An obvious way is to devise a set of benchmarks and put different implementations through their paces. The middleware that results in the fastest computation could be declared as a winner. For this, one would need a standard well defined benchmark which would represent a wide set of financial algorithms, for instance those based on MC methods, and could also generate enough load on the middleware in

[1]http://www.cs.wisc.edu/condor/

[2]http://www.globus.org/

[3]http://glite.web.cern.ch/glite/

[4]http://proactive.inria.fr/

[5]http://sourceforge.net/projects/alchemi/

[6]http://www.unicore.eu/

[7]http://kaapi.gforge.inria.fr/

test.

Benchmarks provide a commonly accepted basis of performance evaluation of software components. Performance analysis and benchmarking, however, is relatively young area in grid computing compared to benchmarks designed for evaluating computer architecture. Traditionally, performance of parallel computer systems has been evaluated by strategically creating benchmark induced load on the system. Typically, such benchmarks comprise of codes, workloads that may represent varied computations and are developed with different programming paradigms. Some examples are Linpack[8], the most popular NAS Parallel benchmark[9] and MPI Benchmarks[10]. The Parkbench, the Linpack and NAS benchmarks are the most widely known benchmarks in the HPC domain and have been used to rank the parallel supercomputers. The MPI benchmarks are used to accurately estimate the execution time of MPI communication operations as used within parallel applications. Since these benchmarks are widely used in ranking the highly parallel systems, a key issue however, is whether these benchmarks can be used "as is" for the grid settings. The adoption of these benchmarks may raise several fundamental questions about their applicability, and ways of interpreting the results. Furthermore, in order to have fair evaluation, any benchmark would have to account for heterogeneity of resources, presence of virtual organizations and their diverse resource access policies, dynamicity due to inherent shared nature of the grid. Such issues in turn have led to broader implications upon methodologies used behind evaluating middleware as discussed in [3, 120]. In Tsouloupas and Dikaiakos (2003) [120], the authors presented GridBench, a tool for benchmarking grid systems. This tool provides in fact a framework for both running benchmarks on a grid and then collecting, archiving and displaying results. It allows integrating any new code to be executed as a benchmark, following a Plug-in like approach.

In our work, however, we are interested in definition of a benchmark in the context of computational finance area, named "SuperQuant" instead of more classical ones coming from scientific computing area such as the NAS benchmark suite. We also implemented a simple tool for running such a benchmark in a grid environment. The "SuperQuant" benchmark mainly solves the option pricing problem in finance. On running this benchmark in a grid, we can indirectly evaluate the performance of financial applications, achievable scalability, ease of deployment across large number of heterogeneous resources and their efficient utilization. Perspectively, we could want to integrate the SuperQuant benchmark into the GridBench presented in [120].

[8]http://www.netlib.org/benchmark/

[9]http://www.nas.nasa.gov/Resources/Software/npb.html

[10]http://hcl.ucd.ie/project/mpiblib

To sum up, this chapter describes the design and development of the SuperQuant Financial Benchmark Suite, a tool to help users investigate various aspects of usage of grid middleware using as simple as possible benchmark kernels. The availability of such kernels can enable the characterization of factors that affect application performance, the quantitative evaluation and comparison of different middlewares. This is done by regarding features as ease of deployment, scalability in resource acquisition through the execution of financial algorithms instead of traditional scientific ones, devised for parallel computing.

4.2 SuperQuant Financial Benchmark Suite

4.2.1 Motivation

In order to produce verifiable, reproducible and objectively comparable results, any middleware benchmark must follow the general rules of scientific experimentation. Such tools must provide a way of conducting reproducible experiments to evaluate performance metrics objectively, and to interpret benchmark results in a desirable context. The financial application developer should be able to generate metrics that quantify the performance capacity of grid middleware through measurements of deployability, scalability, and computational capacity etc. Such metrics can provide a basis for performance tuning of application and of the middleware. Alternatively, the middleware providers could utilize such benchmarks to make necessary specific software design changes. Hence, in order to formalize efforts to design and evaluate any grid middleware, we designed a financial benchmark suite named SuperQuant.

4.2.2 Desired Properties

Some other considerations for the development of this benchmark are described below and significantly follow the design guidelines of NAS benchmarks suite [12].

- Benchmarks must be conceptually simple and easy to understand for both financial and grid computing community.

- Benchmarks must be "generic" and should not favour any specific middleware. Many grid middlewares provide different high level programming constructs such as tailored APIs or inbuilt functionalities like provision for parallel random number generators, etc.

- The correctness of results and performance figures must be easily verifiable. This requirement implies that both input and output data sets must be limited and well defined. Since we target financial applications, we also need to consider real world trading and computation scenarios and data involved therewith. The problem has to be specified in sufficient detail and the required output has to be brief yet detailed enough to certify that the problem has been solved correctly.

- The problem size and runtime requirements must be easily adjustable to accommodate new middleware or systems with different functionalities. The problem size should be large enough to generate considerable amount of computation and communication. In the kernel presented in this chapter, we primarily focus on the computational load while future benchmark kernels may impose communication as well as data volume loads.

- The benchmarks must be readily redistributable.

The financial engineer implementing the benchmarks to be supported by a given grid middleware is expected to solve the problem in the most appropriate way for the given computing infrastructure. The choice of APIs, algorithms, parallel random number generators, benchmark processing strategies, resource allocation is left open to the discretion of this engineer. The languages used for programming financial systems are mostly C, C++ and Java. Most of the grid middlewares are available in these languages and the application developers are free to utilize language constructs that give the best performance possible or any other requirements imposed by the business decisions, on the particular infrastructure available at their organization.

4.3 Components of SuperQuant Financial Benchmark Suite

Our benchmark suite consists of three major components:

- An embarrassingly parallel kernel

- The input/output data and grid metric descriptors

- An output evaluator

Each of these components are briefly described in the following sections.

4.3.1 Embarrassingly Parallel Kernel

We have devised a relatively "simple" kernel which consists of a batch of high dimensional European and barrier options. The objective is to compute price and Greeks values of a number of options with acceptable accuracy and within definite time interval using MC based methods.

Denote the option payoff $\Psi\big(f(S_t), t\big)$ where $S = \{S^1, \ldots, S^d\}$ is a basket of d assets price and $f(\cdot)$ is given by the option's payoff type (e.g. Arithmetic Average, Maximum, or Minimum whose definition can be found in common financial engineering textbooks [61, 126]). Here let us briefly recall the description of European and barrier options:

- Basket European option is a contract that pays $\Psi(f(S_T), T)$ at maturity date T. Hence we run the trajectory realization of S_T directly from Equation (1.20).

- Barrier option is an option that is either activated or cancelled based on the underlying asset price hitting a certain barrier B. For example, a barrier put *down and in* option pays $\Psi(f(S_T), T)\, \mathbb{1}_{\big(\min\limits_{0<t\leq T}(S_t) \leq B\big)}$ at time T. Such option will be activated once the underlying asset prices in the basket hits the barrier B. To price such option using MC methods, we discretize the option life time into a finite set of dates $\Theta = \{t_m = h \times m, m = 1, \ldots, N\}$, with $t_N = T$ and uniform step $h = \dfrac{T}{N}$. One willing to approximate the true price can let m increase to infinity. The trajectory realization of $\{S_{t_m}, t_m \in \Theta\}$ is obtained by solving Equation (1.19).

In the first section of Chapter 1, we provided an example of pricing and hedging a call Geometric Average European option of d assets including the definitions, method description, pseudocodes of pricing and hedging algorithms. Similarly, the European options with other payoff types and barrier options can be priced and hedged using the same pseudo codes with only slight modifications. The full implementation and the exemplary parallel versions of MC based pricing method for such pricing and hedging problems are provided along with the benchmark suite and are available on our website [11].

4.3.1.1 The Composition of the Benchmark Kernel

The core benchmark kernel consists of a batch of 1000 well calibrated *TestCase*s. Each *TestCase* is a multi–dimensional European option with up to 100 underlying assets with necessary attributes like spot prices, payoffs types, time to maturity, volatility, and other

[11]http://www-sop.inria.fr/oasis/plugtests2008/ProActiveMonteCarloPricingContest.html

market parameters. In order to constitute an option, the underlying assets are chosen from a pool of 250 companies listed in the equity S&P500 index[12], while volatility of each asset and its dividend rate are taken from CBOE. The complexity of computation of each

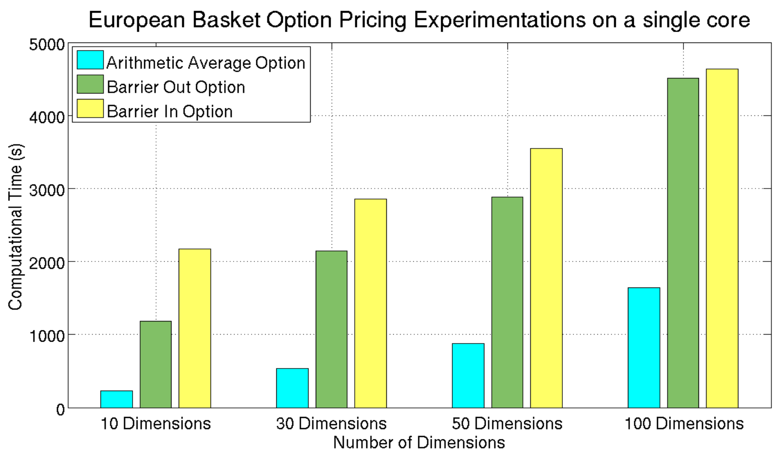

Figure 4.1: Computational time of several European basket option pricings on a single core (Intel Xeon(R), E5335, 2.00GHz, 2G RAM, JDK 1.6)

testcase depends on both the number of dimensions and the length of maturity date. In order to balance the computational time between testcases, we decided that the one with small number of dimensions will have a big maturity and the composition of the batch is as follows:

- 500 *TestCase*s of 10–dimensional European options with 2 years time to maturity
- 240 *TestCase*s of 30–dimensional European options with 9 months time to maturity
- 240 *TestCase*s of 50–dimensional European options with 6 months time to maturity
- 20 *TestCase*s of 100–dimensional European options with 3 months time to maturity

Besides, in practice the options based on up to 10 underlying assets are the most widely traded. Hence in order to be realistic, we distributed half of the number of testcases to 10 dimensional problems. We have 240 examples of both 30 and 50 dimensional testcases and only 20 examples of 100 dimensional ones. Figure 4.1 presents the computational time on

[12]http://www2.standardandpoors.com

a single core for each type of option within the benchmark suite. Thus, the objective of the benchmark is pricing and hedging of a maximum number of *TestCases* in a definite amount of time by implementing the associated algorithms using a given grid middleware.

4.3.1.2 Financial Model Description

As mentioned above, the benchmark kernel includes 1000 multi-dimensional European option contracts with up to 100 underlying assets $S = \{S^1, \ldots, S^{100}\}$ chosen from a pool of 250 companies. Such options are evaluated using the BS model described in (1.2). In Chapter 1, we mentioned the model for simulating these correlated assets by completing the model with a correlation matrix $\big(\rho_{ij},\, i,j = 1,\ldots,d\big)$ of Brownian Motion B, such that $\rho_{ij} = \frac{\mathbb{E}(B_t^i, B_t^j)}{t}$ so $Covariance(B_t^i, B_t^j) = \rho_{ij} t$. Hence the $d \times d$ covariance matrix Cov is defined by,

$$Cov_{ij} = \sigma_i \sigma_j \rho_{ij},$$

where $(\sigma_i, i = 1,\ldots,d)$ is the volatility vector of S and Cov is always a positive-definite matrix. In the next paragraph, we will introduce how to achieve the computation of ρ_{ij}.

Consider a pool of d asset prices, $\big(S_t^i, i = 1,\ldots,d\big)$ continuous in time t within the BS model. We need to compute the correlation matrix $\big(\rho_{ij}\big)$. First we compute from historical data the return value of an asset S^i over a time scale Δt (e.g. a business day or less if we have enough data),

$$X_i(t) = \log\big(S_{t+\Delta t}^i - S_t^i\big).$$

Here the increment $\big(S_{t+\Delta t}^i - S_t^i\big)$ is supposed to be independent of t. We then define a normalized return as

$$x_i(t) = \frac{X_i(t) - \langle X_i \rangle}{\sigma_i}$$

where $\sigma_i = \sqrt{\langle X_i^2 \rangle - \langle X_i \rangle^2}$ is the standard deviation of X_i and $\langle X_i \rangle = \dfrac{1}{N-1} \sum_{n=1}^{N-1} X_i(n)$, where N is the number of observations of S^i during the time t. Then the correlation matrix ρ is constructed as

$$\rho_{ij} = \langle x_i(t) x_j(t) \rangle = \frac{\mathbb{E}(X_i X_j) - \mathbb{E}(X_i)\mathbb{E}(X_j)}{\big(\mathbb{E}(X_i^2) - \mathbb{E}(X_i)^2\big)^{\frac{1}{2}} \big(\mathbb{E}(X_j^2) - \mathbb{E}(X_j)^2\big)^{\frac{1}{2}}}, \tag{4.1}$$

where the expectations above are approximated by the Law of Large Number (LLN) rule, such that $\mathbb{E}(X_i) \simeq \dfrac{1}{N-1} \sum_{n=1}^{N-1} X_i(n)$. By this construction, all the coefficients of ρ are restricted to the interval $[-1, 1]$. Since the coefficient $\rho_{ij} = \langle x_i(t) x_j(t) \rangle$, in matrix notation, such matrix ρ can also be expressed as

$$\rho = \frac{1}{N}\mathbb{X}\mathbb{X}^t \tag{4.2}$$

where $\mathbb{X}$ is a $d \times N$ matrix with elements $\big(x_{i,n} \equiv x_i(n \times \Delta t); i = 1, \dots, d; n = 1, \dots, N\big)$. However, such historical correlation matrix ρ is not always able to be used directly in any financial application (e.g. option pricing, portfolio optimization) if the number of observations N is not very large compared to d. Let us give an example: if we construct a historical correlation matrix of first 250 assets in the S&P500 index list using 254 observations (e.g. from 07–Avril–2008 to 07–Avril–2009, it means 1 year data, $N/d = 1.016$), the result is a non positive–definite matrix which can not be directly used in an option pricing application. However, if we increase the number of observations to 505 for the same 250 assets (e.g. from 07–Avril–2007 to 07–Avril–2009, it means 2 year data, $N/d = 2.02$), the result is a positive–definite one. Therefore in the first case where we can not increase the number of observations by any reason, we need to re-calibrate the historical correlation matrix in order to make it applicable for the financial applications. We do recalibration as follows :

We diagonalize the historical correlation matrix ρ such that $\rho = VDV^t$ where V is the matrix that contains the eigenvectors of ρ and D is the diagonal matrix whose diagonal contains the eigenvalues of ρ. We set a small non–negative value to the negative diagonal elements of D. Then we compute the matrix $\overline{D}$ which is the reduction matrix of D by simplifying the eigenvalues of D within the interval $\big[\lambda_{R,min}, \lambda_{R,max}\big]$, as follow

$$\overline{D} = \sum_{j=1}^{n} D^j + \sum_{j=d-m+1}^{d} D^j + diag(\underbrace{0, \dots, 0}_{n\ elements}, T_D, T_D, \dots, T_D, \underbrace{0, \dots, 0}_{m\ elements}) \tag{4.3}$$

where $D^j = diag(\underbrace{0, \dots, 0, \lambda_{\rho,j}, 0, \dots, 0}_{d\ elements})$ and the constant T_D is the trace of the matrix D. The two bounded numbers $\big(\lambda_{R,min}, \lambda_{R,max}\big)$ are computed using the random matrix theory, see more details in Laloux (2000) [70] and Plerou and al. (2000) [96]. To be more clear, each D^j is a square matrix of order d, with the elements of eigenvector on the main diagonal. Once having the new diagonal matrix $\overline{D}$, we reconstruct an approximation of the historical correlation matrix ρ under the new form $\overline{\rho} = V\overline{D}V^\top$. The last step consists in re-normalizing the coefficients of the matrix $\overline{\rho}$ such that

$$\overline{\rho}_{ij} = \frac{\overline{\rho}_{ij}}{\sqrt{\overline{\rho}_{ii}}\sqrt{\overline{\rho}_{jj}}} \tag{4.4}$$

Now, the matrix $\overline{\rho}$ presents a well–defined correlation matrix that is a positive–definite matrix with the diagonal equal to 1 and all the coefficients are in the interval of $[-1, 1]$.

4.3.2 Input/Output Data and Grid Metrics Format

To facilitate processing, exchanging and archiving of input data, output data and grid related metrics, we define relevant XML data descriptors. The *TestCases* required by the kernel and the "reference" results are also included in the benchmark suite.

- Input ***AssetPool :*** represents the database of 250 assets required to construct a basket (collection) option of assets

- Input ***CorrelationMatrix :*** defines a correlation matrix of each testcase. Such provided matrix is positive-definite with diagonal values 1 and correlation coefficients in the interval of $[-1, 1]$ and ready to use. The calibration of a historical correlation matrix was described above in the previous section.

- Input ***TestCases :*** defines a set of *TestCase*s, input parameters, needed by the pricing and hedging algorithm discussed above. Each *TestCase* includes parameters such as an option, which is a subset of *AssetPool*, a sub-matrix of *CorrelationMatrix*, type of payoff, type of option, barrier value if needed, interest rate, maturity date, etc.

- Output ***Results :*** defines a set of *Result* which consists of *Price* and *Greeks* values of each individual *TestCase* and time *Metrics* required to compute each output value.

- Output grid ***Metrics :*** is defined to store the total time required for the entire computation.

4.3.3 Output Evaluator

The output evaluator is a tool to compare the results computed by different implementations of the benchmark kernel *TestCases* with "reference" results provided in the suite. The values we provide as "reference" results are approximated using Monte Carlo simulations because we can not obtain them by the analytical methods. So for them to be able to be "reference", we need to rely on a very high number of simulations. Hence such results are obtained within an acceptable confidence interval. Since these prices are not obtained by an analytical method, we call them the "reference" prices, using " " !

4.3.3.1 Evaluation Criteria

Based on the Central Limit Theorem of MC methods mentioned in Section 1, the estimated option price V is obtained with a 95% confidence within the following interval $\left[\pm 1.96\frac{\widehat{s}_C}{\sqrt{nbMC}}\right]$, where the estimator $\widehat{s}_C$ computes the standard deviation of V and $nbMC$ is the number of MC simulations, see Equation (1.1.2).

We decide the tolerable relative error in computing the results is 10^{-3} which required a number of MC simulations of $\frac{10^6}{\widehat{s}_C}$. Since the accuracy of the computed results relies on the spot prices of the underlying assets, we consider relative errors with respect to the "reference" results (see below). These "reference" results are computed in advance (e.g. in the definition phase of the benchmark suite) with sufficiently large number of MC simulations in order to achieve lower confidence interval as briefly explained above, see more in the next section.

The **Output Evaluator** employs a point based scheme to grade the results and also provides a detailed analysis of points gained per *TestCase*. Thus we have outlined the following points based system:

- The main evaluation criteria is the total number of finished testcases, say M, that are priced during the assigned time slot. For each price computed, we have +10 points. Thus at maximum, we can earn up to $+10 \times M$ points.

- If the computed price is within the expected precision, we gain +5 points

- If the computed price is above the expected precision, we gain +10 points

- If the computed price is below the expected precision, we are penalized with -10 points.

- For each Greek letter, namely Delta, Gamma, Rho and Theta that is precisely computed, we will get +2 points per Greek letter. The Greek letters must be computed by a finite difference method with a fixed step size. Note that if non–precise, the values will not be given any points.

- For each minute saved out of the assigned time slot, then we gain +1 point.

4.3.3.2 "Reference" Results Validation

The "reference" results provided in the benchmark suite are not analytical results and are computed by using MC based methods. It is well known in BS model that there only exists analytical solution for European option pricing in case of one dimension otherwise in case of multi-dimension we have to use other approximation approaches such as MC based methods.

We observed that in some very specific cases we can analytically reduce a basket of assets into a one-dimensional “reduced” asset. Further details of the reduction technique were given in the third section of Chapter 1. The exact option price on this “reduced” asset can be computed by using BS formula. Thus in such particular cases, we can validate the “reference” results based on the relative error compared to the exact results. These “reference” results will be validated once they achieve equal or better accuracy than the tolerable relative error 10^{-3} which we have decided, otherwise we must calibrate the number of MC simulations until they satisfy such condition. This calibration helps us to have an idea about a large enough number of MC simulations for computing other “reference” results especially for those that do not have any analytical solution for comparison. Such calibration has to be done before releasing the benchmark suite. To highlight the usefulness of this approach, we provide below a numerical example.

Numerical Example: Consider a call/put GA option of 100 independent assets ($d = 100$) with prices modeled by SDEs (1.2). The parameters are given as $S_0^i = 100, i = 1, \dots, 100$, $K = 100$, $r = 0.0$, $\delta_i = 0.0$, $\sigma_i = 0.2$ and $T = 1$ year. The basket option is simulated by using 10^6 MC simulations by using Algorithm 1. The “reduced” asset is $\Sigma_t = \prod_{i=1}^{d} S_t^{i\,\frac{1}{d}}, i = 1, \dots, 100$ and it is the solution of the one–dimensional SDE: $d\Sigma_t/\Sigma_t = \left(\widetilde{\mu}dt + \widetilde{\sigma}dZ_t\right)$ where $\widetilde{\mu} = r + \frac{\sigma^2}{2d} - \frac{\sigma^2}{2}, \widetilde{\sigma} = \frac{\sigma}{\sqrt{d}}$ and Z_t is a Brownian Motion. The parameters of Σ are given as $\Sigma_0 = 100, \widetilde{\mu} = 0.0198, \widetilde{\sigma} = 0.02$. We are interested in comparing the estimated option price V of d assets with the analytical “reduced” one $\widetilde{V}$ on Σ. The computation of $\widetilde{V}$ was described in Section 1.3 of Chapter 1. We denote the absolute error $\Delta V = |V - \widetilde{V}|$, then the relative error η is computed as $\eta = \frac{\Delta V}{\widetilde{V}}$.

In Table 4.1, the first column represents the estimated option prices using MC methods and their 95% confidence interval. The second column gives the analytical exact option prices. The last two columns show the absolute and relative errors. As it can be observed, the relative error in case of put option pricing is less than 10^{-3}, therefore we validate such put option price as reference. Meanwhile the call price is not, thus to validate the call option price we have to increase the number of MC simulations. The Table 4.2 shows that such call option price is validated whenever using 10^8 MC simulations. Based on this calibration analysis, we can find, for each “reference” result in the benchmark suite, an optimal number of MC simulations which can produce a relative error less than 10^{-3} (e.g. for any call option pricing on up to 100 underlying assets, we should consider at least 10^8 MC simulations).

In this example, we also consider the Delta hedging for both MC based option pricing

Table 4.1: Call/Put price of a GA of 100 assets option using 10^6 MC simulations and of the "reduced" option

Call MC Price V (95% CI)	"Reduced" Call Price $\widetilde{V}$	Absolute error	Relative error
0.16815 (0.00104)	0.16777	3.8×10^{-4}	2.2×10^{-3}
Put MC Price V (95% CI)	"Reduced" Put Price $\widetilde{V}$	Absolute error	Relative error
2.12868 (0.00331)	2.12855	1.3×10^{-4}	6.1×10^{-5}

Table 4.2: Call price of a GA of 100 assets option using 10^8 MC simulations and of the "reduced" option

Call MC Price V (95% CI)	"Reduced" Call Price $\widetilde{V}$	Absolute error	Relative error
0.16793 (0.00011)	0.16777	1.6×10^{-4}	9.5×10^{-4}

Table 4.3: Delta values of the Call/Put GA of 100 assets and of the "reduced" one

Basket MC Call Delta Δ^i	"Reduced" Call Delta $\widetilde{\Delta}$
0.00160	0.16030
Basket MC Put Delta Δ^i	"Reduced" Put Delta $\widetilde{\Delta}$
-0.00818	-0.82010

and analytical option pricing. In the first case, the Delta hedging produces a vector of 100 first order derivatives of the option price $\left(\Delta^i, i = 1, \ldots, 100\right)$ with respect to the change of each asset price among 100 assets. Following the initial parameters, every asset has the same spot price and volatility rate, hence Δ^i, $\forall i$ are uniform. Such Delta values are computed by using finite difference methods as described in Algorithm 2. In the later case, the Delta value $\widetilde{\Delta}$ is computed by using an analytical formula [61]. The relation between Δ^i and $\widetilde{\Delta}$ is described as follows,

$$\prod_{i=1}^{d} \Delta^i = \left(\frac{1}{d}\widetilde{\Delta}\right)^d. \tag{4.5}$$

Further details of this relation are given in Section 1.3 of Chapter 1. In Table 4.3, we present the numerical results of the Delta hedging for both high dimensional and reduced option. As we can observe, since the option prices are validated, the Delta values of both options follow the relation in Equation (4.5). As illustrated by such numerical example, we are able to validate our "reference" results in the benchmark suite and also to verify the option pricing implementation.

4.4 Proof of Concept : The V grid Plugtest and Contest

As a proof of concept, we used the SuperQuant Benchmark Suite for the **2008 SuperQuant Monte Carlo Challenge** organized as part of **V GRID Plugtest and Contest**[13] at INRIA Sophia Antipolis. The details of the contest and the benchmark input data can be found on the Plugtest and Contest website[14].

4.4.1 Test-Bed

Each contest participant was given an exclusive one hour grid access for evaluating the benchmark on two academic grids, Grid'5000[15] and InTrigger[16], which combined consisted of around 5000 computational cores geographically distributed across France and Japan. The description of Grid'5000 resources which were provided during the contest is given in Table 4.4.

4.4.2 ProActive Monte Carlo API

In order to provide a simple programming framework in a distributed environment for the Grid Plugtest participants, we developed what is named the ProActive Monte Carlo API (MC API), based on the underlying ProActive Master/Worker framework [17]. This MC API aims to ease the running of Monte Carlo simulations on the grid through the ProActive grid middleware and its programming APIs and is well suited for embarrassingly parallel problems (as the parallel European option pricing using MC methods which constitutes kernel of the SuperQuant benchmark suite). In such a quite simple case, it is enough to create a master and a group of workers: then tasks have simply to be distributed uniformly to workers and merged later by the master. Master, worker creation and task creation and distribution are in fact handled by the Master/Worker ProActive API. The MC API is simply an extension of Master/Worker ProActive API for problems which would need MC simulations to be solved in parallel. The main features of the ProActive MC API are as follows:

[13]http://www.etsi.org/plugtests/GRID2008/VGRID_PLUGTESTS.htm

[14]http://www-sop.inria.fr/oasis/plugtests2008/ProActiveMonteCarloPricingContest.html

[15]https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

[16]https://www.intrigger.jp/wiki/index.php/InTrigger

[17]http://proactive.inria.fr/release-doc/ProActive/api_published/index.html

Table 4.4: Grid’5000 configuration for the “2008 SuperQuant Monte Carlo” challenge. Total number of machines is 1244

Location	# of machines	CPU	# of CPUs	OS	Gflops	JVM
Grid’5000 Bordeaux	48	Opteron 248	2	Fedora Core 4	19.4	Sun 1.5.0_08 amd64
Grid’5000 Lille	15	Opteron 252	2	Red Hat EL3	29.7	Sun 1.5.0_08 amd64
	53	Opteron 248	2	Red Hat EL3	29.7	Sun 1.5.0_08 amd64
Grid’5000 Lyon	56	Opteron 246	2	Debian 3.1	20.9	Sun 1.5.0_08 amd64
	70	Opteron 250	2	Debian 3.1	14.6	Sun 1.5.0_08 amd64
Grid’5000 Nancy	47	Opteron 246	2	Debian testing	15.6	Sun 1.5.0_08 amd64
Grid’5000 Orsay	216	Opteron 246	2	Debian testing	unspecified	Sun 1.5.0_08 amd64
	126	Opteron 250	2	Debian testing	unspecified	Sun 1.5.0_08 amd64
Grid’5000 Rennes	64	Xeon IA32 2.4Ghz	2	Debian testing	unspecified	Sun 1.5.0_08 i586
	99	Opteron 246	2	Debian testing	unspecified	Sun 1.5.0_08 amd64
	64	Opteron 248	2	Debian 3.1	26.6	Sun 1.5.0_08 amd64
	64	G5 2Ghz	2	OS X	unspecified	Sun 1.5.0_06 PowerPC
Grid’5000 Sophia	105	Opteron 246	2	Rocks Linux	36.5	Sun 1.5.0_08 amd64
	56	Opteron 275	2	Rock Linux	25.9	Sun 1.5.0_08 amd64
Grid’5000 Toulouse	58	Opteron 248	2	Fedora Core 3	unspecified	Sun 1.5.0_08 amd64
Grid’5000 Grenoble	103	Itanium 2 1Ghz	2	Red Hat EL3	unspecified	Sun 1.5.0_08 i586

- It integrates a mechanism for generating parallel random number sequences using the SSJ package written by Pierre l'Ecuyer. The relevant discussion of using the SSJ package for this MC based problem were mentioned earlier in Chapter 2. Each worker has its own independent sequence of random numbers and there is a guarantee that two different workers, will generate sequences that won't overlap until a certain number of experiences (2^{127}).

- Monte Carlo simulations or other tasks can be run on remote workers. Tasks are defined by implementing an interface, named *EngineTask*.

- A small set of common numerical operations required for computational finance are developed (e.g. Geometric Brownian Motion, the Brownian Bridge, etc).

Using the MC API, the deployment of the workers infrastructure is hidden to programmers and is done through GCM deployment descriptors (see Chapter 2). Hence programmers only have to focus on the application implementation.

In this section we will illustrate how to use the MC API for developing distributed European option pricing. The programming must address the following steps:

1. Initialization : First, programmers create a runner class, say *EuropeanOption* and initialize a root *PAMonteCarlo* that references the deployment descriptor and two virtual node names. That root has the responsibility to handle master/worker creation and deployment, task distribution, results collection.

   ```
   // In the runner class EuropeanOption
   PAMonteCarlo mcRoot = new PAMonteCarlo
           (URL descriptorURL, String workersVNName, String masterVNName);
   ```

   For master creation, the API has the possibility to create either a local master (on the processor the runner class is running on) or a remote master. The workers deployment triggered by the MC API relies on the use of ProActive Master/Worker framework deployment mechanism. Regardless of the way it is created any active object instantiation is transparent to the MC API programmers. For more detailed explanation of the deployment mechanism and of ProActive deployment descriptors, see Chapter 2.

2. Tasks Definition and Submission :

   Before creating task, programmers have to create a class, say *MCEuropeanBasketOption*, for basket European option pricing. Such *MCEuropeanBasketOption* class contains the reference to *mcRoot* and the market information requested for the pricing problem. Hence it has the responsibility to create and distribute tasks. Notice that task implementation will be managed by programmers.

```
// In the runner class EuropeanOption
MCEuropeanBasketOption euroBasketOption =
                new MCEuropeanBasketOption(mcRoot, basketParams,
                         testCaseCorrMatrix, testCases, testCaseAssetPool);
// Tasks implementation
MCEuropeanPricingEngineTask euroBasketOptionTask =
        new MCEuropeanPricingEngineTask();
// Task setting
euroBasketOption.setPricingEngine(euroBasketOptionTask);
// Task execution by calling the built-in run() method implicitly
euroBasketOption.calculateMC();
```

The method *calculateMC* computes the price of the option using MC methods.

```
public void calculateMC() throws ProActiveException, TaskException {
        result = (Result)mcRoot.run(pricingEngine);
        result.setTestCaseID(pricingEngine.getTestCase().getTestCaseID());
}
```

The *run(·)* method implemented within *MCEuropeanPricingEngineTask* class will create and launch a bag of tasks of MC simulations for computing the European option price implemented in *MCEuropeanPricingAlgorithm* class. The *MCEuropeanPricingAlgorithm* class will perform a number of Monte Carlo simulations (which is predefined as the task size by the user). Each simulation provides a value and *MCEuropeanPricingAlgorithm* class then returns the sum of such values. The details of such implementation can be found on the Plugtest and Contest website[18]. The sum value from the individiual tasks in the bag are collected and averaged for computing the final price of the European option.

```
// In the class MCEuropeanPricingEngineTask
public class MCEuropeanPricingEngine extends AbstractPricingEngine {
// Launches the bag of Monte Carlo simulations.
        public Result run(final Simulator simulator) {
                final List<SimulationSet<Result>> set = new
                ArrayList<SimulationSet<Result>>(DEFAULT_NUMBER_TASKS);
```

[18]http://www-sop.inria.fr/oasis/plugtests2008/ProActiveMonteCarloPricingContest.html

```
                Enumeration<Result> simulatedPriceList = null;
                for (long i = 0; i < DEFAULT_NUMBER_TASKS; i++) {
                        set.add(new MCEuropeanPricingAlgorithm
                        (basketParams, testCase, assetPool,
                        correlationMatrix,DEFAULT_MCITER));
                }
                // Submitting this set of tasks.
                try {
                            simulatedPriceList = simulator.solve(set);
                } catch (final TaskException e) {
                            throw new RuntimeException(e);
                }
                final ResultHelper finalResultHelper = new ResultHelper();
                // collect individual results of tasks
                while (simulatedPriceList.hasMoreElements()) {
                        final Result partialResults
                        = simulatedPriceList.nextElement();
                        // accumulate all the results
                        finalResultHelper.update(partialResults);
                }
                // Finalizing the accumulated prices, this requires the total
                // number of simulations for the algorithm.
                finalResultHelper.finalizePrices(DEFAULT_NUMBER_TASKS,
                        DEFAULT_MCITER, testCase.getOption(),
                        basketParams, assetPool, testCase);
                return finalResultHelper.getResult();
}
```

3. Results Gathering : Results are collected by the master when the calculations are complete. There are two ways of waiting for the results. These two ways strongly rely on the ProActive Master/Worker API. The users can either wait until one or every result is available (the thread blocks until the results are available) or ask the master for result availability and continue until the results are finally available. In the second case the application thread does not block while the results are computed.

   ```
   // In the runner class EuropeanOption
   Result result = euroBasketOption.getResult();
   ```

4. Resources Releasing : At the end, the root uses one single method to terminate the master and also terminates the workers manager as well, thus eventually freeing every

resources. A boolean parameter tells the master to free resources or not (i.e. terminate remote JVMs).

```
// In the runner class EuropeanOption
mcRoot.terminate();
```

4.4.3 Challenge Results

Figure 4.2 presents the final results of the Challenge. The participants primarily used

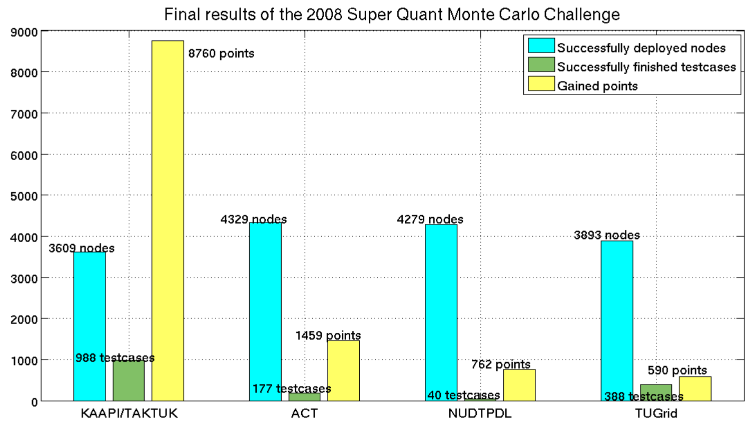

Figure 4.2: Final results of the 2008 SuperQuant Monte Carlo challenge

two middlewares, ProActive and KAAPI/TAKTUK, which couples KAAPI, a Parallel Programming Kernel and TAKTUK, a middleware for adaptive deployment. Both middleware implement the Grid Component Model (GCM) deployment model, recently standardized by the ETSI GRID[19] technical committee for deploying the application over a large number of grid nodes [16]. The infrastructure descriptors and application descriptors required by GCM deployment were bundled with the benchmark suite.

As we can see in Figure 4.2, the KAAPI/TAKTUK team was successful in computing 98.8% of total number of *TestCases* and was also able to deploy application on a significantly large number of nodes. The other teams used ProActive to implement the benchmark kernel. From Figure 4.2, we can observe that the benchmarks were not only useful to quantitatively compare two middleware solutions, but also gave the opportunity to evaluate different benchmark implementations using the same middleware. Such comparison is useful not only to middleware providers but also to grid application developers.

[19]http://www.etsi.org/WebSite/Technologies/GRID.aspx

However, since the ProActive MC API relies on the one master/multiple workers pattern implemented within the ProActive Master/Worker API, the teams who used it faced a poor performance due to the overhead in communication between too many workers and the master. To avoid this, the solution is to adopt an alternative architecture of one master/multiple sub-masters/multiple workers where each sub-master only handles a not too large group of workers. Such a more scalable architecture has been discussed in Chapter 2.

4.5 Conclusions

In this chapter we have presented SuperQuant Financial Benchmark Suite for performance evaluation and analysis of grid middleware in the financial engineering context. We described the preliminary guidelines for designing the benchmark. We then described the benchmark constituents along with a brief overview of the embarrassingly parallel benchmark kernel. Now as this benchmark kernel is defined, it could be used within other benchmarking tools (e.g. the Gridbench tool [120]). As a proof of concept, we also utilized this benchmark in a grid programming contest. Within the contest context, in order to greatly simplify the grid aspects in application implementation for participants, we provided a simple programming framework, named ProActive Monte Carlo API. Such MC API eases running Monte Carlo simulations on the grid and is well suited for embarrassingly parallel problems. However, since it is simply based on the single level ProActive Master/Worker framework, it raised some limitations when used on large-scale heterogeneous infrastructures as grids, so its concrete use to implement the SuperQuant benchmark suite was not really successful. Nevertheless, the MC API might be relevant to be used by itself and hence is promoted as one component of the ProActive Parallel Suite [20].

[20]http://proactive.inria.fr

Chapter 5

CONCLUSIONS AND PERSPECTIVES

This research work was initiated and took place in the context of the French “ANR-CIGC GCPMF” project (Grilles de Calcul Pour les Mathématiques Financières) with the aim to show the efficiency of grid computing in performing the Monte Carlo based intensive calculations in financial derivative pricing applications. Applying parallel computing for computational finance is a very active research and development area, and it is still a very promising research domain.

In Chapter 2, we presented PicsouGrid, a grid computing framework for Monte Carlo based financial applications. PicsouGrid provides fault tolerance, load balancing, dynamic task distribution and deployment on a heterogeneous grid environment. PicsouGrid has been successfully deployed on the multi-site French Grid’5000 for European option pricing. We achieved very good performances for such European option pricing experiments, using up to 190 processors on a cluster and up to 120 processors on a 4-sites grid. Fault-tolerance and load balancing are realized transparently for the programmer, based on processor replacement and dynamic and aggressive load balancing. PicsouGrid also proved a very adaptable tool for experimenting parallel solutions to the high dimensional American option pricing problems which are computational intensive in their serial form.

In Chapter 3 we particularly proposed a parallelization of the Classification Monte Carlo (CMC) algorithm, originally devised by Picazo (2002) [94], for high dimensional American option pricing. Thanks to the use of PicsouGrid, we were able to run this algorithm for pricing American options requiring important computation times, because of a large number of opportunity dates (e.g. up to 100 such dates), and because of large dimension (e.g. up to 40 underlying assets). We also compared and analyzed the option price accuracy obtained with this CMC algorithm when using different classification algorithms such as AdaBoost, Gradient Boost and Support Vector Machines. Beside, we provided a full explanation and some detailled numerical experiments of the dimension reduction technique which is very useful in validating the estimated option prices using CMC algorithm.

Additionally in Chapter 4, we define a financial benchmark suite for performance evaluation and analysis of grid computing middlewares. Such benchmark suite is conceptually simple and easy to understand for both the grid computing and financial communities. The benchmark suite was successfully used in the 2008 SuperQuant Monte Carlo challenge - the

Fifth Grid Plugtest and Contest. Within the context of this challenge, we developed the ProActive Monte Carlo API (MC API), now integral part of the ProActive Parallel Suite.

In general this research work provided a view of applying grid computing for finance particularly in computational intensive high dimensional option pricing, thus issuing some perspectives. PicsouGrid framework, presented in Chapter 2, is successfully used in case of high dimensional European and American option pricing. In future, we aim to use PicsouGrid in exploring other financial computational intensive applications such as portfolio management, Value at Risk computation, etc. On the other hand, the PicsouGrid scope is not restricted within the financial domain but should also be well suited for accelarating any application using Monte Carlo methods in other domains such as biomolecular simulations, protein structure prediction, etc. Of course, such collaboration would be a very promising work in the future. Returning to our thesis context, using the parallel approach of the CMC algorithm, we can reduce significatively the computational time for pricing large dimensional American options. However, these performance results are still far from a longer term goal which aims to price these options within a critical time (i.e. within 1 minute). Hence other alternative parallelization strategies for the entire CMC algorithm, described in Chapter 3 should be realized. Besides, parallel version of Support Vector Machines [54, 28], of Boosting [50, 72, 78] should be designed and implemented to spare additional time in the classification step (even if its sequential duration is proportionally small compared to the other steps) for the next parallel version of CMC algorithm. Obvisouly, such parallel CMC algorithm (as a kernel with complex communication) could be integrated in the SuperQuant benchmark suite in order to truly understand the overhead imposed by grid middleware in financial applications.

BIBLIOGRAPHY

[1] L.A. Abbas-Turki and B. Lapeyre. American Options Pricing on Multi-Core Graphic Cards. *International Conference on Business Intelligence and Financial Engineering*, pages 307–311, 2009.

[2] Y. Achdou and O. Pironneau. *Computational methods for option pricing.* Society for Industrial Mathematics, 2005.

[3] P. Alexius, B.M. Elahi, F. Hedman, P. Mucci, G. Netzer, and Z.A. Shah. A Black-Box Approach to Performance Analysis of Grid Middleware. *Lecture Notes in Computer Science*, 4854:page 62, 2008.

[4] F. Ametrano and L. Ballabio. Quantlib-a free/open-source library for quantitative finance. *www.datasynapse.com*, 2003.

[5] L. Andersen and M. Broadie. Primal-Dual Simulation Algorithm for Pricing Multidimensional American Options. *Management Science*, 2004.

[6] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society Washington, DC, USA, 2004.

[7] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[8] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. *Lecture Notes in Computer Science*, 2862:61–86, 2003.

[9] S. Kumar A.R. Choudhury, A. King and Y. Sabharwal. Optimizations in Financial Engineering: The Least-Squares Monte Carlo method of Longstaff and Schwartz. *Parallel and Distributed Computing in Finance Workshop, IPDPS*, 2008.

[10] L. Baduel, F. Baude, and D. Caromel. Asynchronous typed object groups for grid programming. *International Journal of Parallel Programming*, 35(6):573–614, 2007.

[11] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, composing, deploying for the grid. *Grid Computing: Software Environments and Tools. Springer, Heidelberg*, 2005.

[12] D. Baileym, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. *Technical Report RNR-91-002 Revision 2, NAS Systems Division, NASA Ames Research Center*, August 1991.

[13] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, T. Kielmann, J. Maassen, R. van Nieuwpoort, J. Romein, L. Renambot, T. Ruhl, et al. The distributed ASCI supercomputer project. *ACM SIGOPS Operating Systems Review*, 34(4):96, 2000.

[14] V. Bally, G. Pagès, and J. Printems. A quantization tree method for pricing and hedging multidimensional American options. *Mathematical Finance*, 15(1):119–168, 2005.

[15] J. Barraquand and D. Martineau. Numerical valuation of high dimensional multivariate American securities. *Journal of Financial and Quantitative Analysis*, pages 383–405, 1995.

[16] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM: A grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1):5–24, 2009.

[17] R. Bellman. *Adaptive control processes: a guided tour. 1961.* Princeton University Press.

[18] S. Bezzine, V. Galtier, S. Vialle, F. Baude, M. Bossy, V.D. Doan, and L. Henrio. A Fault Tolerant and Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing in Finance. *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[19] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.

[20] G.E.P. Box and M.E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, pages 610–611, 1958.

[21] P. Boyle, M. Broadie, and P. Glasserman. Monte Carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21(8-9):1267–1321, 1997.

[22] P. Boyle, J. Evnine, and S. Gibbs. Numerical evaluation of multivariate contingent claims. *The Review of Financial Studies*, 2(2):241–250, 1989.

[23] P. Bratley and H. Niederfieiter. Implementation and tests of low-discrepancy sequences. *ACM Transactions on Modeling and Computer Simulation*, 2(3):195–213, 1992.

[24] M. Broadie and J. Detemple. The Valuation of American Options on Multiple Assets. *Mathematical Finance*, 7(3):241–286, 1997.

[25] M. Broadie and P. Glasserman. Pricing American-style securities using simulation. *Journal of Economic Dynamics and Control*, 21(8-9):1323–1352, 1997.

[26] M. Broadie and P. Glasserman. A stochastic mesh method for pricing high-dimensional American options. *Journal of computational finance*, 7:35–72, 2004.

[27] A.L. Bronstein, G. Pagès, and J. Portès. Multi-asset American options and parallel quantization. *http://hal.archives-ouvertes.fr/hal-00320199/PDF/PAO.pdf*, 2008.

[28] L.J. Cao, SS Keerthi, C.J. Ong, JQ Zhang, U. Periyathamby, X.J. Fu, and HP Lee. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks*, 17(4):1039, 2006.

[29] D. Caromel, C. Delbe, A. Di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[30] J.F. Carriere. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance Mathematics and Economics*, 19(1):19–30, 1996.

[31] J.P. Chancelier, B. Lapeyre, and J. Lelong. Using Premia and Nsp for constructing a risk management benchmark for testing parallel architecture. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. IEEE Computer Society, 2009.

[32] S.K. Chaudhary. American options and the LSM algorithm: Quasi-random sequences and Brownian bridges. *Journal of Computational Finance*, 8(4), 2005.

[33] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, FAB Silva, CO Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416, 2003.

[34] E. Clément, D. Lamberton, and P. Protter. An analysis of a least squares regression method for American option pricing. *Finance and Stochastics*, 6(4):449–471, 2002.

[35] P.D. Coddington and A.J. Newell. JAPARA–A Java Parallel Random Number Generator Library for High-Performance Computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, New Mexico*. Citeseer, 2004.

[36] Platform Computing Corporation. LSF Version 4.1 Administrators Guide. *Platform Computing, www.platform.com*, 2001.

[37] J.C. Cox, S.A. Ross, and M. Rubinstein. Option Pricing: A Simplified Approach. *Journal of Financial Economics*, 1979.

[38] J.C. Cox and M. Rubinstein. *Options markets.* Prentice Hall, 1985.

[39] N. Cristianini and J. Shawe-Taylor. *An introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[40] V.D. Doan, A. Gaikwad, F. Baude, and M. Bossy. "Gridifying" classification-Monte Carlo algorithm for pricing high-dimensional Bermudan-American options. In *High Performance Computational Finance Workshop, Supercomputing 2008*, 2008.

[41] V.D. Doan, A. Gaikwad, M. Bossy, F. Baude, and F. Abergel. A financial engineering benchmark for performance analysis of grid middlewares. Technical Report 0365, INRIA, 2009.

[42] V.D. Doan, A. Gaikwad, M. Bossy, F. Baude, and I. Stokes-Rees. Parallel pricing algorithms for multi–dimensional bermudan/american options using monte carlo methods. Research Report 6530, INRIA, 2008.

[43] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of parallel computing.* Morgan Kaufmann Los Altos, CA, USA, 2007.

[44] A. Foster and C. Kesselman. The GRID: Blueprint for a new computer infrastructure, 1998.

[45] Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. *Proceedings of the Thirteenth International Conference in Machine Learning*, 1996.

[46] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *Ann. Statist*, 2000.

[47] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[48] M.C. Fu and J.Q. Hu. Sensitivity analysis for Monte Carlo simulation of option pricing. *Probability in the Engineering and Informational Sciences*, 9(3):417–446, 1995.

[49] M.C. Fu, S.B. Laprise, D.B. Madan, Y. Su, and R. Wu. Pricing American options: A comparison of Monte Carlo simulation approaches. *Journal of Computational Finance*, 4(3):39–88, 2001.

[50] V. Galtier, O. Pietquin, and S. Vialle. AdaBoost Parallelization on PC Clusters with Virtual Shared Memory for Fast Feature Selection. In *IEEE International Conference on Signal Processing and Communications, 2007. ICSPC 2007*, pages 165–168, 2007.

[51] N.C.D. Gelernter and N. Carriero. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[52] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer, 2004.

[53] T. Glatard, J. Montagnat, and X. Pennec. Medical image registration algorithms assessment: Bronze standard application enactment on grids using the MOTEUR workflow engine. *Studies in health technology and informatics*, 120:93, 2006.

[54] H.P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The cascade svm. *Advances in neural information processing systems*, 17(521-528):2, 2005.

[55] J. Gustafson. ClearSpeed Technology Inc. *private communication.*

[56] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA Data Mining Software: An Update. Volume 11, 2009.

[57] D. Harris. Will financial services drive Grid adoption. *Grid Today*, 3:40, 2004.

[58] M.B. Haugh and L. Kogan. Pricing American options: a duality approach. *Operations Research*, pages 258–270, 2004.

[59] R. Hochreiter, C. Wiesinger, and D. Wozabal. Large-scale computational finance applications on the open grid service environment. *Lecture notes in computer science*, 3470:891, 2005.

[60] K. Huang and R.K. Thulasiram. Parallel Algorithm for Pricing American Asian Options with Multi-Dimensional Assets. *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, 2005.

[61] J. Hull. *Options, futures, and other derivatives.* Pearson Prentice Hall, 2008.

[62] J. Hull and A. White. The use of the control variate technique in option pricing. *Journal of Financial and Quantitative Analysis*, 23(3):237–251, 1988.

[63] A. Ibanez and F. Zapatero. Monte Carlo Valuation of American Options through Computation of the Optimal Exercise Frontier. *Journal of Financial and Quantitative Analysis*, 2004.

[64] P. Jackel and P. Jaeckel. *Monte Carlo methods in finance.* John Wiley & Sons, 2002.

[65] N. Ju. Pricing by American option by approximating its early exercise boundary as a multipiece exponential function. *Review of Financial Studies*, 11(3):627–646, 1998.

[66] D.E. Knuth. The art of computer programming: Seminumerical algorithms, volume 2, 1981.

[67] E. Korn. *Option pricing and portfolio optimization: modern methods of financial mathematics.* Amer Mathematical Society, 2001.

[68] H.W. Kuhn and A.W. Tucker. Nonlinear programming. *ACM SIGMAP Bulletin*, pages 6–18, 1982.

[69] T.L. Lai and F. AitSahalia. Exercise boundaries and efficient approximations to American option prices and hedge parameters. *J. Computational Finance*, 4:85–103, 2001.

[70] L. Laloux, P. Cizeau, M. Potters, and J.P. Bouchaud. Random matrix theory and financial correlations. *International Journal of Theoretical and Applied Finance*, 3(3):391–398, 2000.

[71] D. Lamberton and B. Lapeyre. *Introduction to stochastic calculus applied to finance.* CRC Press, 1996.

[72] A. Lazarevic and Z. Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed and parallel databases*, 11(2):203–229, 2002.

[73] P. L'Ecuyer. Random number generation. *Handbook of simulation: principles, methodology, advances, applications, and practice*, 1998.

[74] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: SSJ: a framework for stochastic simulation in Java. *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, 2002.

[75] P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proceedings of the 32nd conference on Winter simulation*, pages 683–689. Society for Computer Simulation International San Diego, CA, USA, 2000.

[76] C. Lemieux. Randomized quasi-Monte Carlo: A tool for improving the efficiency of simulations in finance. In *Simulation Conference, 2004. Proceedings of the 2004 Winter*, volume 2, 2004.

[77] F.A. Longstaff and E.S. Schwartz. Valuing American options by simulation: a simple least-squares approach. *Review of Financial Studies*, 2001.

[78] F. Lozano and P. Rangel. Algorithms for parallel boosting. In *Machine Learning and Applications, 2005. Proceedings. Fourth International Conference on*, page 6, 2005.

[79] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. *ACM Sigplan notices*, 34(8):173–182, 1999.

[80] Constantinos Makassikis, Stéphane Vialle, and Xavier Warin. Large Scale Distribution of Stochastic Control Algorithms for Financial Applications. In *The First Workshop on Parallel and Distributed Computing in Finance (Computational Finance) PDCoF08*, pages 1–8, 04 2008.

[81] H. Markowitz. Portfolio selection. *Journal of finance*, pages 77–91, 1952.

[82] M. Mascagni. Some methods of parallel pseudorandom number generation. *IMA Volumes in Mathematics and Its Applications*, 105:277–288, 1999.

[83] M. Mascagni and A. Srinivasan. SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software-TOMS*, 26(3):436, 2000.

[84] L. Mason, J. Baxter, P. Bartlett, and M. Frean. Boosting algorithms as gradient descent in function space. In *Proc. NIPS*, volume 12, pages 512–518. Citeseer, 1999.

[85] N. Meinshausen and B.M. Hambly. Monte Carlo methods for the valuation of multiple-exercise options. *Mathematical Finance*, 14(4):557–583, 2004.

[86] R.C. Merton. Theory of rational option pricing. *The Bell Journal of Economics and Management Science*, pages 141–183, 1973.

[87] D. Meyer, F. Leisch, and K. Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169–186, 2003.

[88] J. Montagnat, D. Jouvenot, C. Pera, A. Frohner, P. Kunszt, B. Koblitz, N. Santos, and C. Loomis. Bridging clinical information systems and grid middleware: a Medical Data Manager. *Studies in health technology and informatics*, 120:14, 2006.

[89] M. Montero and A. Kohatsu-Higa. Malliavin Calculus applied to finance. *Physica A: Statistical Mechanics and its Applications*, 320:548–570, 2003.

[90] M. Monteyne and R.M. Inc. RapidMind Multi-Core Development Platform. *RapidMind, Tech. Rep*, 2007.

[91] M. Moreno and J.F. Navas. On the robustness of least-squares Monte Carlo (LSM) for pricing American derivatives. *Review of Derivatives Research*, 6(2):107–128, 2003.

[92] I. Muni Toke and J.-Y. Girard. Monte carlo valuation of multidimensional american options through grid computing. In Springer, editor, *Lecture Notes in Computer Science : Proc. Large Scale Scientific Computation (LSSC 05)*, 2005.

[93] G.C. Pflug, A. Swiketanowski, E. Dockner, and H. Moritsch. The AURORA financial management system: model and parallel implementation design. *Annals of Operations Research*, 99(1):189–206, 2000.

[94] J.A. Picazo. American Option Pricing: A Classification-Monte Carlo (CMC) Approach. *Monte Carlo and Quasi-Monte Carlo Methods 2000: Proceedings of a Conference Held at Hong Kong Baptist University, Hong Kong SAR, China, November 27-December 1, 2000*, 2002.

[95] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. *Advances in Kernel Methods-Support Vector Learning*, 1999.

[96] V. Plerou, P. Gopikrishnan, B. Rosenow, L.A.N. Amaral, and HE Stanley. A random matrix theory approach to financial cross-correlations. *Physica A: Statistical Mechanics and its Applications*, 287(3-4):374–382, 2000.

[97] C. Reisinger and G. Wittum. On multigrid for anisotropic equations and variational inequalities Pricing multi-dimensional European and American options. *Computing and Visualization in Science*, 7(3):189–197, 2004.

[98] M. Roehrig, W. Ziegler, and P. Wieder. GFD-I.11: Grid Scheduling Dictionary of Terms and Keywords. Technical report, Global Grid Forum, 2002. Grid Scheduling Dictionary Working Group.

[99] L.C.G. Rogers. Monte Carlo valuation of American options. *Mathematical Finance*, 2002.

[100] M. Rubinstein and E. Reiner. Breaking down the barriers. *Risk*, 4(8):28–35, 1991.

[101] P.A. Samuelson. Proof that properly anticipated prices fluctuate randomly. *Management Review*, 6(2), 1965.

[102] R.E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[103] B. Scholkopf, C.J.C. Burges, and A.J. Smola. *Advances in kernel methods: support vector learning*. The MIT press, 1998.

[104] E.S. Schwartz. The valuation of warrants: implementing a new approach. *Journal of Financial Economics*, 4(1):79–93, 1977.

[105] S.E. Shreve. *Stochastic Calculus for Finance: Continuous-Time Models*. Springer, 2004.

[106] M. Sordo and Q. Zeng. On sample size and classification accuracy: A performance comparison. *Lecture Notes in Computer Science*, 3745:193, 2005.

[107] D. Steinberg and P. Colla. CARTclassification and regression trees. *San Diego, CA: Salford Systems*, 7, 1997.

[108] L. Stentoft. Convergence of the least squares Monte Carlo approach to American option valuation. *Management Science*, pages 1193–1203, 2004.

[109] W.R. Stevens et al. *Advanced programming in the UNIX environment*. Addison-Wesley, 1992.

[110] I. Stokes-Rees, F. Baude, V.D. Doan, and M. Bossy. Managing parallel and distributed Monte Carlo simulations for computational finance in a grid environment. *Proceedings of the International Symposium on Grid Computing in 2007*, 2009.

[111] Data Synapse. The Next Generation of High Performance Computing For Financial Services. *White Paper, www.datasynapse.com*, 2001.

[112] A.C. Tan and D. Gilbert. Ensemble machine learning on gene expression data for cancer classification. *Applied Bioinformatics*, 2:75–84, 2003.

[113] D. Tavella. *Quantitative methods in derivatives pricing: an introduction to computational finance*. Wiley, 2002.

[114] D. Tavella and C. Randall. *Pricing Financial Instruments: The Finite Difference Method*. Wiley, New York, 2000.

[115] Digipede Technologies. DIGIPEDE: Grid Computing for Windows. *www.digipede.net*.

[116] R.K. Thulasiram and D.A. Bondarenko. Performance Evaluation of Parallel Algorithms for Pricing Multidimensional Financial Derivatives. *The International Conference on Parallel Processing Workshops (ICPPW02)*, 2002.

[117] C.Y. Tilley. *Interpretative archaeology*. Berg Publishers, 1993.

[118] M. Tomek. *A stochastic tree approach to pricing multidimensional American options*. University of London, 2006.

[119] J.N. Tsitsiklis and B. Van Roy. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks*, 12(4):694–703, 2001.

[120] G. Tsouloupas and MD Dikaiakos. Gridbench: A tool for benchmarking grids. In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 60–67, 2003.

[121] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[122] R.V. Van Nieuwpoort, J. Maassen, T. Kielmann, and H.E. Bal. Satin: Simple and efficient java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, 2005.

[123] S. Villeneuve. Exercise regions of American options on several assets. *Finance and Stochastics*, 3(3):295–322, 1999.

[124] J.W.L. Wan, K. Lai, A.W. Kolkiewicz, and K.S. Tan. A parallel quasi-Monte Carlo approach to pricing multidimensional American options. *International Journal of High Performance Computing and Networking*, 4(5):321–330, 2006.

[125] C. Wiesinger, D. Giczi, and R. Hochreiter. An Open Grid Service Environment for large-scale computational finance modeling systems. *LNCS*, 3036:83–90, 2004.

[126] P. Wilmott, J. Dewynne, and S. Howison. *Option pricing: mathematical models and computation.* Oxford financial press, 1993.

[127] P. Wilmott et al. *Derivatives: The theory and practice of financial engineering.* Wiley, 1998.

[128] S.A. Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25(13-14):2149–2175, 1999.