HIGH PERFORMANCE MONTE CARLO BASED OPTION PRICING ON ENTERPRISE GRIDS

Qusay F. Hassan

Faculty of Computers and Information Systems, Mansoura University, Mansoura, Egypt

***ABSTRACT*:** The Monte Carlo method is being increasingly used in various fields like mathematical finance, engineering, physical sciences, and bioinformatics to solve problems where using the deterministic method is infeasible. The Monte Carlo simulation is a numerical computational technique which uses thousands or even millions of random values to solve complex problems, causing this technique to be slow and computer-intensive when being used for options pricing. Therefore, financial firms are usually forced to deploy powerful hardware means such as supercomputers and computer clusters in order to perform the needed simulations. The ability to link the traditional computers and servers available at organizations to form a grid that acts as a supercomputer can enable both scientists and professionals to run their simulations without spending extra costs. This paper proposes and implements an application for the Monte Carlo methods for options pricing using the enterprise grids on the Windows environment. The paper also provides a comparison between the performance of the proposed framework and the traditional model.

***KEYWORDS*:** Financial Analysis, Monte Carlo Simulation, Option Pricing, Grid Computing, Enterprise Grids, Alchemi, .NET Framework

1 Introduction

The Monte Carlo method is a numerical computational technique that is used to solve problems by means of repeated random samplings [And86]. This technique is being widely used to solve problems that are infeasible with the deterministic algorithm. The Monte Carlo method enables scientists to simulate systems and scenarios using thousands or even millions of randomly generated inputs.

The Monte Carlo simulations are used in finance, mathematical finance and financial engineering to value the financial assets with various sources of uncertainty or when no practical close form solutions are available. Phelim P. Boyle was the first scientist to apply the Monte Carlo methods in finance in order to calculate the value of the European options [Boy77].

In general, the accuracy of the Monte Carlo results is affected by the law of large numbers where the convergence equals $\frac{1}{\sqrt{n}}$ and n is the number of random simulations. In other words, halving the error in the generated results requires increasing the number of sampled values by a factor of four. This indicates that simulating one scenario using the Monte Carlo method at an acceptable accuracy rate might increase the processing time dramatically. This drawback restricts the use of the Monte Carlo simulations on powerful computer clusters and expensive supercomputers [TBG08]. However, one of the good characteristics of the Monte Carlo method is parallelism, where multiple paths can be computed simultaneously.

Grid computing is a form of parallelization that is being increasingly used in different fields to complete slow and computationally-intensive tasks in short times. For simulations in general and for the Monte Carlo simulations in specific, the goal of cutting the simulation time can be efficiently achieved by linking a (large) pool of computers to calculate independent paths concurrently. Forming grids out of traditional desktop machines and laptops (and sometimes servers and clusters) available at the enterprises is known as Enterprise Grids or Desktop Grids. This paper proposes a framework that utilizes enterprise grids to speed up the Monte Carlo simulations for European options pricing. The remainder of this paper is organized as follows: Section 2 gives a historical review for the related work. Section 3 introduces the basic terms of options and option pricing. The fundamentals, definition and architecture of grid computing and enterprise grids are briefly discussed in section 4. In section 5, the architecture and technical details of our implementation for the Monte Carlo-based options pricing on enterprise grids are presented. Experimental results and evaluation of our framework are discussed in section 6. Finally, section 7 concludes and hints at some of the future work.

2 Related Work

Different studies were conducted to parallelize the analysis and the calculation of financial models. Both hardware and software-based models were utilized to achieve this goal efficiently.

For the hardware-based model, some papers discussed the use of computer clusters with off-the-shelf accelerators to perform the Monte Carlo simulations for options pricing [TBG08]. Although this model is usually faster than the software-based parallelization models, most organizations cannot afford it. Furthermore, such clusters are usually designed and tailored to execute specific tasks making them far more complex, less configurable, less customizable, and less flexible in terms of the range of the problems that they can solve.

Graphics processing units (GPUs) which are used to accelerate the building of images and videos offer processing powers that sometimes surpass the power of traditional CPUs. Some studies discussed options pricing and Monte Carlo option pricing on GPUs including work presented by NVIDIA Corporation [KP05]. These studies showed the substantial speedup gained from the use of GPUs over the traditional CPUs. However, this approach is limited by the availability of GPUs on enterprises' computers and by the new programming skills that are required to write GPU-based applications. The installation of GPUs is usually limited to gaming workstations and computers used to render 3D models and animations. Also, programming using the GPU model is still in its early stages and most developers find it difficult to use.

For the software parallelization model, Message Passing Interface (MPI) was proposed as a method where the calculation phase is subdivided into several processes and each process is executed on a separate processor (or a set of processor) [ABM01]. Although this approach can distribute the work across multiple processors, it is not as flexible to the changing conditions as grid computing. Moreover, parallelizing the financial analysis calculations and simulations using MPI incurs a lot of time and effort as it requires rewriting applications, taking into consideration that implementers are responsible for writing the brokering and scheduling logic.

Some work explored the use of the Monte Carlo method and global grids in the field of financial services [M+07]. Global grids are different from the approach proposed in this paper in that it connects resources offered by different organizations (and hosted on different platforms) by forming different administrative organizational units known as virtual organizations (VOs) [FKT01]. Although global grids are scalable and autonomous, they are harder and much more complex to build as they require extensive coordination and synchronization between different organizations, higher security measures, and integration between the resources that are hosted on various hardware architectures/platforms and operation systems. Global grids are also more suitable for long-running tasks where each task may take several hours to complete.

3 The Basics of Option Pricing

This section introduces the definitions and basic terms related to options and option pricing.

3.1 Options

An option or financial option is a contract (also known as a security) between two parties that specifies the terms of quantity, price, conditions (including the right to buy or sell), class, and the expiration date of the underlying asset. The underlying asset can be a stock, currency, debt, index, bond, real estate, commodity, and the like.

Different styles of options are available such as Exotic, Non-vanilla, and most commonly American and European options. The option's style refers to the class to which the option belongs. In general, the date on which an option can be exercised is the most important factor that defines the option's style. Exercising an option means that the holder is able to buy or sell the underlying asset, according to the type of the option, terminating the contract between the two parties is immediate. For example, American options can be exercised at any time up to their expiration date, whereas European options can only be exercised at their expiration date. This paper focuses on the valuation of European options.

Before delving into the details of the options pricing and the grid-based implementation, the readers should first understand the basic terms of an option [***11a]:

- *Spot Price:* The current market price of the underlying asset.
- *Strike Price:* The specified price on the option contract.
- *Maturity:* The expiration date of the contract at which the option can be exercised.
- *Volatility:* The relative change of the option price. The volatility value is generally affected by the market's conditions and the option's supply and demand.
- *Risk-free Rate:* In finance, this variable refers to the return rate an investor expects to make from a risk-free investment over a given period of time. This assumption is theoretical since no investment is free of risk or financial loss and thus, risk-free rate can be estimated by the variance in the actual return around the expected return.

There are two main types of options [Hul11]:

1. *Call Options:* An option that gives its holder the right (not an obligation) to buy the underlying asset by a certain date for a certain price. If K is the strike price and ST is the spot price, then the payoff of European call is calculated using equation (1).

$$S(T) = \max(ST - K, 0) \quad (1)$$

2. *Put Options:* An option that gives its holder the right (not an obligation) to *sell* the underlying asset by a certain date for a certain price. The payoff European put is calculated using equation (2).

$$S(T) = \max(K - ST, 0) \quad (2)$$

3.2 Monte Carlo Technique for Option Pricing

As illustrated in Fig. 1, the idea behind the pricing of financial options using the Monte Carlo method is straightforward where the valuation process is broken down into four simple steps [Gla03].

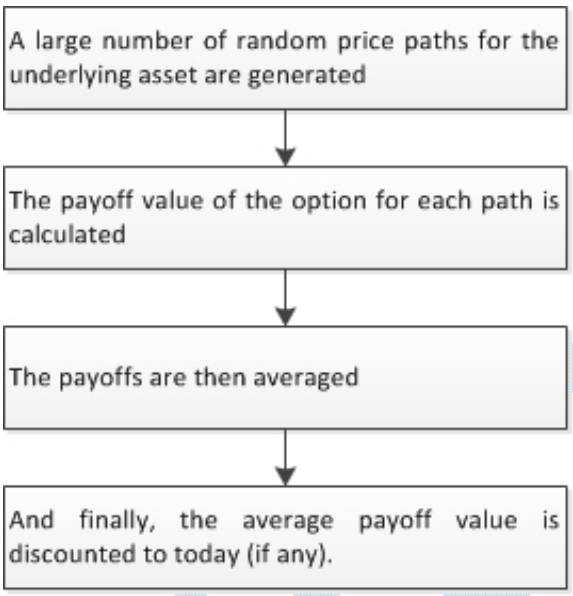

Fig. 1. Steps of Monte Carlo-based options pricing

In order to implement a Monte Carlo simulation, a simulation for the geometric Brownian motion (GBM) process for the underlying asset is needed [Ros09]. Based on the Black-Scholes model, the evolution of the stochastic differential model (SDE) can be determined using equation (3) [FS73].

$$dS = \mu S dt + \sigma S dWt \quad (3)$$

The change in the option price is *dS*; μ is the drift rate; *σ* is the volatility; *Wt* is a standard Brownian motion; *dt* is a time interval/increment at *t* = 0; and the value of *S(0)* is *S0* which represents the current value of the option.
Based on equation (3), the solution of the SDE is derived as illustrated in equation (4).

$$S(T) = S(0)\exp\left(\left[r - \frac{1}{2}\sigma^2\right]\right)T + \sigma W(T)) \quad (4)$$

The risk-free interest rate (which in general is continuously compounded) is *r*; and *W(T)* is a random variable. Based on the standard normal distribution of *W(T)*, equation (4) can be re-written as illustrated in equation (5).

$$S(T) = S(0)\exp\left(\left[r - \frac{1}{2}\sigma^2\right]\right)T + \sigma\sqrt{T}Z) \quad (5)$$

Z is a random variable of the standard normal distribution (mean 0 and deviation 1).
To valuate an option with the Monte Carlo method, a large number of simulations is required: a minimum of 10,000 simulations is needed to price an option with a satisfactory accuracy. This constraint makes pricing with the Monte Carlo method slow and computer-intensive.
As mentioned earlier, the standard error in the estimated value is mainly related to the square root of the number of simulations. Thus, quadrupling the number of simulations will halve the error. For instance, at least 40,000 simulations are needed to double the accuracy of an option price resulting from 10,000 simulations. Certainly, this will make the valuation process much slower and more computer-intensive. From equations (4) and (5), equations (6) and (7), which are used to calculate the European call price and put price, can be deduced, respectively.

$$C = \frac{1}{n} e^{-rT}\left(\sum_{i=1}^{n} \max(S(T) - \mathrm{k}, 0)\right) \quad (6)$$

$$P = \frac{1}{n} e^{-rT}\left(\sum_{i=1}^{n} \max(\mathrm{K} - S(T), 0)\right) \quad (7)$$

The discount factor is e^{-rT}; and n is the number of random simulations.
To obtain more accurate results from the Monte Carlo–based option pricing, the simulations can be performed in a path-dependent manner. This form, which will be used for the evaluation of our framework, simply means that the simulation process is divided into a set of discrete time steps where the option price will be evaluated in each time step in addition to the master evaluation illustrated in equations (7) and (8). If *m* is the number of time steps, then equations (8) and (9) can be used for the evaluation of the path-dependent European call price and European put price, respectively.

$$C = \frac{1}{n}\, e^{-rT}\left(\sum_{i=1}^{n} \sum_{j=1}^{m} \max\left(S_{j-1}(T) - \mathrm{k}, 0\right)\right) \quad \textbf{(8)}$$

$$P = \frac{1}{n}\, e^{-rT}\left(\sum_{i=1}^{n} \sum_{j=1}^{m} \max\left(\mathrm{K} - S_{j-1}(T), 0\right)\right) \quad \textbf{(9)}$$

4 Grid Computing

Grid computing refers to the ability to combine, coordinate and share different computing resources in a scalable and dynamic manner to obtain powerful capabilities in order to meet the complicated needs that are hard to cost-effectively fulfill by other means [FK04].

Supercomputers are usually utilized in cases where enormous resources are needed to solve very complex and computationally-intensive problems quickly and efficiently. Examples of these problems include deigning a new plane or vehicle, financial analysis and modeling, weather forecasting, risk analysis, urban planning, pharmaceutical engineering, film making and 3D rendering. However, most organizations, especially small-to-medium enterprises (SMEs) and non-profit organizations cannot afford such solutions. Grid computing allows organizations to have a supercomputer-like performance but at much lower costs. This goal is accomplished by networking these resources in a way that allows end users to use them autonomously as if they are all coming from a single powerful computer.

As mentioned earlier, grid computing is one form of the parallel computing models where processing takes place concurrently on a number of nodes (also known as hosts). These nodes can be standalone machines, servers, clusters, supercomputers, or a combination of any of them. The interaction between grid nodes can take different forms. A Master/Worker parallel model is one of the commonly followed paradigms in grid computing. As its name denotes, this model is composed of two constituent parts: 1) A Master which is a central node (or cluster of nodes) that is responsible for scheduling and dispatching the work units, and receiving the results; 2) Workers that work together to execute the assigned jobs. Grid workers can either be full time workers solely dedicated to executing the incoming jobs, or non-dedicated where they join the grid to voluntarily execute the jobs only when they are idle and then leave once they are needed to perform traditional tasks.

As illustrated in Fig. 2, a user sends a complex job to the master node via a grid-enabled application. This client application is responsible for breaking a complex job into a number of smaller and simpler sub-jobs (also known as subtasks). Then, the master allocates workers in order to assign the queued sub-jobs to. As illustrated in Fig. 2, a user sends a complex job to the master node via a grid-enabled application. This client application is responsible for breaking a complex job into a number of smaller and simpler sub-jobs (also known as subtasks). Then, the master allocates workers in order to assign the queued sub-jobs to. The result of every completed sub-job is then sent back to the master in order to have it forwarded back to the client. According to the client application logic, it can decide whether to use the received results "as is" or to execute some post-processing operations such as formatting the results or combining them into a single output.

Windows operating system (OS) is the most prominent operating system around the world. Different versions of Widows OS are already installed on over 80% of personal computers and corporate servers [***11b]. Thus, deploying grids on a Windows environment is reasonable and could be beneficial for both profit and non-profit enterprises.

.NET Framework (pronounced as "dot net") is the cutting-edge development technology offered by Microsoft for Windows environment. A number of grid technologies compatible with the .NET environment are now available in the market. Some of these products are open-source such as Alchemi [***11c], some are freeware such as Utilify [***11d], and others are commercial such as Aneka [***11e] and DigiPede [***11f].

To implement the proposed framework, Alchemi has been used. Alchemi is a grid-enabling middleware with a set of Application Programmable Interfaces (APIs) developed by the GRIDS research team at Melbourne University [L+05]. Alchemi is composed of three main components: 1) Manager which acts as a master that manages the whole grid. 2) Executor which acts as a grid worker that actually process tasks. 3) Dashboard which enables implementers to monitor the state and utilization of the running grid(s), and the number of running applications and executors.

Alchemi deploys a thread-like programming model where the computational process is subdivided into a set of smaller units that are distributed over the available number of executors [L+05]. The manager queues the created jobs and schedule them according to the availability of the executors. If the number of the queued jobs is larger than that of the free executors, the manager assigns a number of jobs that matches the free executors, and when one of the assigned jobs has been completed, the manager sends a new job to the returning executor, and so on. Alchemi offers three events that fire according to

the status of the grid jobs in order to notify the end users (and developers): 1) "ThreadFinish" which fires when a single grid job is successfully completed. 2) "ThreadFailed" which fires when a grid job has failed. 3) "ApplicationFinish" which fires when all grid jobs are executed.
The latest stable release of Alchemi is 1.0.6 which can be downloaded from the Alchemi page on SourceForge [***11g]. Alchemi 1.06 works fine on Windows XP and Windows 7 and it comes with easy installation wizards. However, a number of difficulties have been encountered in installing and using Alchemi on Windows Vista.

5 Implementation

This section gives the readers all the details regarding the implemented framework including the architecture and the C# code written to grid-enable the Monte Carlo simulation for European Option Pricing.

5.1 Architecture

Fig 3, illustrates the graphical user interface (GUI) of our grid-based Monte Carlo simulator. As mentioned, the framework has been implemented using .NET framework and C# language as well as Alchemi APIs. The end user passes the connection string information to the deployed grid including the grid IP, port number, username and password. The end user also passes the simulation parameters so that the simulator can calculate the assigned tasks. These parameters are: 1) An Excel document containing the simulation parameters/variables; 2) Excel ranges from which the valuation parameter can be extracted; 3) the appropriate sheet; 4) and finally the output range for the returned results. Each column in the spreadsheet represents a specific type of variable (such as spot, strike, etc.); whereas each row represents a set of input variables required to run one valuation process. Thus, the granularity level of our grid jobs is one row (or valuation task) per a job. As will be discussed latter, job sizing is one of the most important factors for the success of the grid-enabling phase for the Monte Carlo simulations.
The readers should note that storing the input parameters is not limited to Excel. Other means such as text files, comma-separated values (CSVs), Extensible Markup Language (XML) and databases can be used to store and pass these parameters, and to save the returned result sets as well. Fig. 4 illustrates a snapshot of some of the input parameters placed in the Excel sheet used for our Monte Carlo simulations.

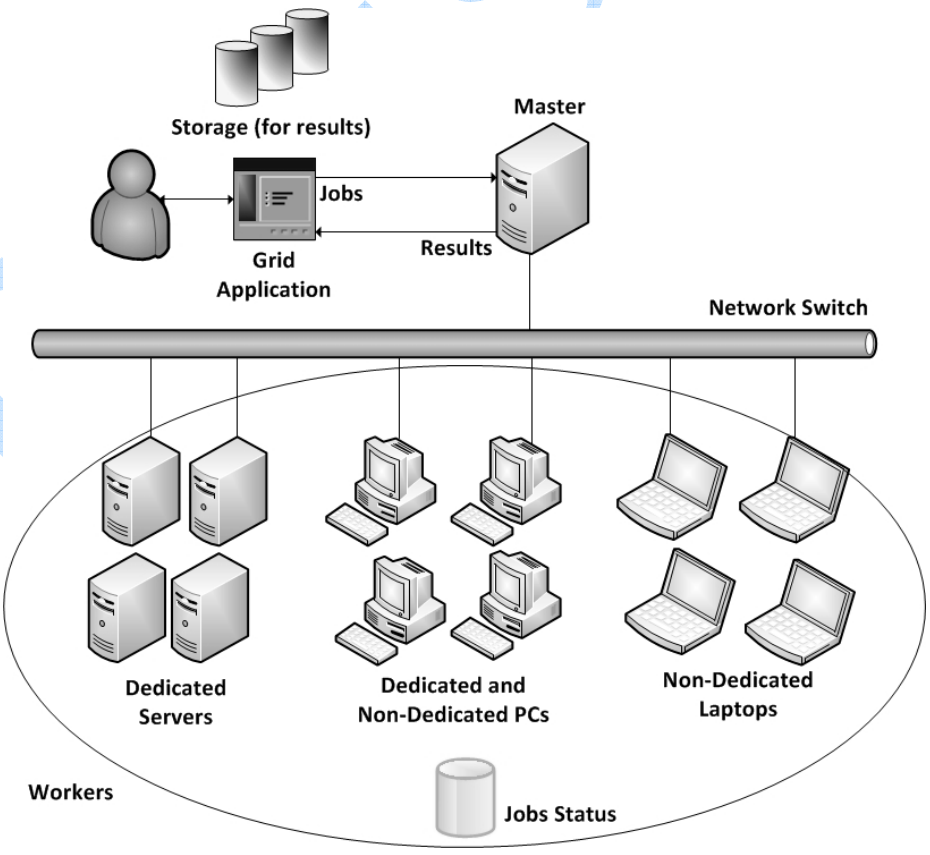

Fig. 2. High level architecture of the grid computing model

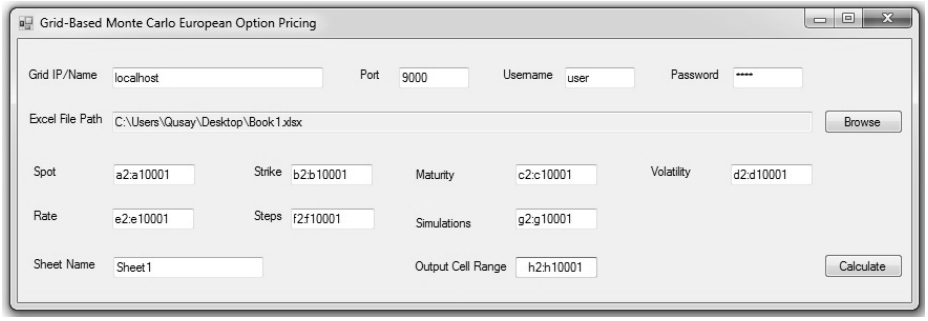

Fig. 3. Client application

	Spot	Strike	Maturity	Volatility	Risk-free Rate	Time Steps	Number of Simulations
1							
2	100	100	1	0.2	0.05	1	100000
3	100	110	1	0.2	0.05	1	100000
4	100	110	1	0.45	0.05	1	100000
5	100	110	3	0.4	0.05	1	100000
6	100	110	3	0.2	0.05	10	100000
7	100	100	1	0.2	0.05	1	100000
8	100	100	1	0.2	0.05	1	100000

Fig. 4. Sample input parameters

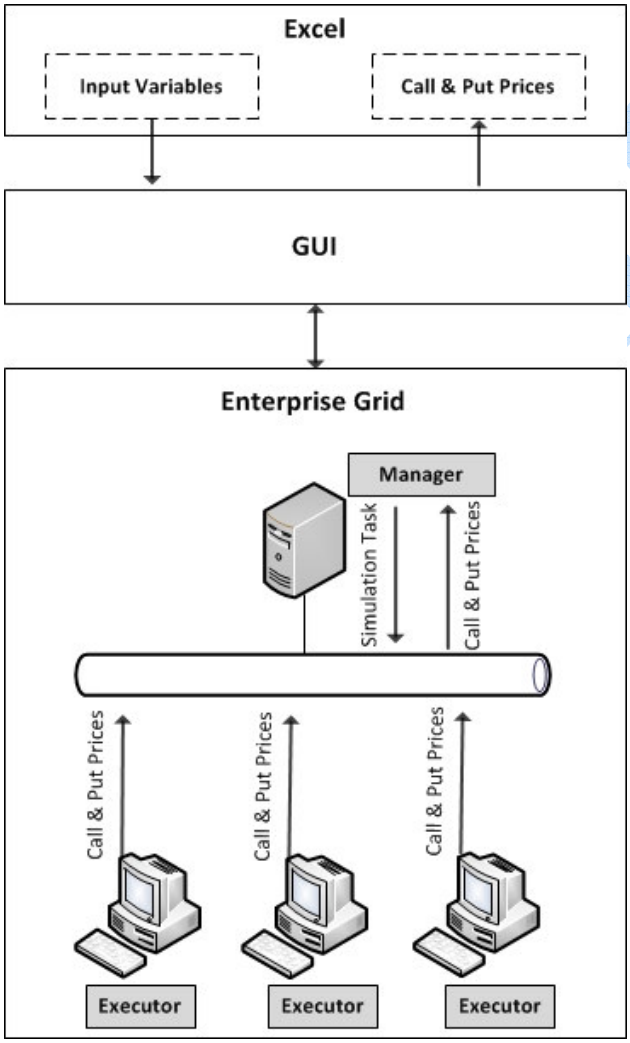

Fig. 5. Monte-Carlo based Options Pricing Framework

As illustrated in Fig. 5, the architecture of our framework is typically based on the general architecture of the enterprise grids (showed in Fig. 2). The end user passes the valuation parameters to the grid through the client application which in turn creates the needed number of grid jobs. When these jobs are received by the manager, they are queued and scheduled for execution on the connected executors. After distributing the jobs to the connected executors, the manager waits till some results are received. When one valuation task is completed or has failed, the end user is notified via the client application by firing the "ThreadCompeted" event or the "ThreadFaild" event, respectively. Developers are able to catch the failed jobs and write a logic that can re-submit them to the grid or process them locally. When all jobs are completed, the client application stores the returned call prices and put prices into the designated column(s).

5.2 Client Code

The code of the client application is responsible for: 1) enabling the end user to feed the grid with the input values (i.e., simulation variables); 2) creating the simulation jobs based on the passed variables and configuration settings. The client application also contains the GUI elements, components and logic. The logic that is used to read user inputs, create the simulation jobs and submit them to the grid is abstracted below:

```
//Connection string to the grid
//Parameters: server name, port number, username, pass-
word
GConnection gc = new GConnection("localhost", 9000,
"user", "user");

//Creates a new grid application
GApplication App = new GApplication(gc);
App.ApplicationName = "European Option Pricing Monte
Carlo Simulation";

//Adds the module containing EuropeanOptionPricing-
Thread to the application manifest
App.Manifest.Add(new ModuleDepend-
ency(typeof(EuropeanOptionPricingThread).Module));

//Load input file (omitted for two reasons 1. each devel-
oper may use different logic; 2. to make code simpler
//Iterate through input value
EuropeanOptionPricingThread
_EuropeanOptionPricingThread = new EuropeanOption-
PricingThread(spot, strike, maturity, volatility, rate, steps,
simulations);
//Adds the created job (thread) to the list of the grid jobs.
These jobs will be distributed to the grid executors for
processing
App.Threads.Add(_EuropeanOptionPricingThread);
//Start processing on the grid
App.Start();
[C# code of the client application]
```

5.3 Grid Enabling Code

As illustrated below, to create grid jobs, a class that implements the "GThread" class offered by Alchemi must be created. "GThread" class is one of the public classes offered by the Alchemi APIs. To implement the "GThread" class, developers must first add a reference to the "Alchemi.Core" dynamic library (.dll) in their code, and then add the "using Alchemi.Core.Owner;" directive in their implementation class. The implementation class must be marked with the "[Serializable]" attribute otherwise runtime errors will be fired and no jobs will be executed on the grid.

```
using System;
using Alchemi.Core.Owner;
namespace OptionPricing
{
   [Serializable]
   public class EuropeanOptionPricingThread : GThread
   {
      double _spotPrice, _strike, _maurity, _volatility,
_riskFreeRate;
      int _nsteps;
      long _nsimulations;
      double[] _result;

public EuropeanOptionPricingThread(double spotPrice,
double strike, double maurity, double volatility, double
riskFreeRate, int nsteps, long nsimulations)
{
   _spotPrice = spotPrice;
   _strike = strike;
   _maurity = maurity;
   _volatility = volatility;
   _riskFreeRate = riskFreeRate;
   _nsteps = nsteps;
   _nsimulations = nsimulations;
}
//This method is responsible for executing the processing
logic on one of the grid nodes (executors)
public override void Start()
{
   //Create an instance of the class that contains the actual
Monte Carlo simulation logic for Option Pricing
   EuropeanOptionPricing _EuropeanOptionPricing =
new EuropeanOptionPricing(_spotPrice, _strike,
_maurity, _volatility, _riskFreeRate, _nsteps,
_nsimulations);
   //Call the simulation method
   _result = _EuropeanOptionPricing.Calculate();
}
}
}
[C# code of that implements the Start() method of Al-
chemi]
```

5.4 Calculation Code

As illustrated below, the calculation/simulation logic is encapsulated in a method called "Calculate()". This method, in basic terms, performs the four Monte Carlo steps illustrated in Fig. 1 and briefly described in equations (3), (8) and (9) [Hau06]. As illustrated, the "Calculate()" method refers to two external methods, namely "GenerateRandomNumbers()" and "GetMean()".
"GenerateRandomNumbers()" is responsible for generating a set of random values that will be used during the simulation process. In this work, a pseudorandom number generator using the Box-Müller transform is used [BM58].
"GetMean()" method is responsible for averaging the values of the generated paths (i.e., payoffs).

```
using System;
namespace OptionPricing
{
   [Serializable]
   public class EuropeanOptionPricing
   {
      //Members
      double _spotPrice, _strike, _maurity, _volatility,
_riskFreeRate, _nsteps, _nsimulations;

      //Constructor
      public EuropeanOptionPricing(double spotPrice,
```

```
double strike, double maurity, double volatility, double
riskFreeRate, double nsteps, double nsimulations)
      {
         _spotPrice = spotPrice;
         _strike = strike;
         _maurity = maurity;
         _volatility = volatility;
         _riskFreeRate = riskFreeRate;
         _nsteps = nsteps;
         _nsimulations = nsimulations;
      }

//Calculates the Call and Put prices and returns them in a
two-elements (single dimension) array
public double[] Calculate()
{
   double dt = _maurity / _nsteps;
   double vsqrdt = _volatility * Math.Pow(dt, 0.5);
   double drift = (_riskFreeRate -Math.Pow(_volatility, 2)
/ 2) * dt;
   double[] callpayoffvec;
   callpayoffvec = new double[(int)_nsimulations];
   double[] putpayoffvec;
   putpayoffvec = new double[(int)_nsimulations];
   long counter = 1;
   double[] randvec = NRandVars(_nsteps *
_nsimulations);
   for (long i = 1; i <= _nsimulations; i++)
   {
      double st = _spotPrice;
    double curtime = 0;
    for (int j = 1; j <= _nsteps; j++)
    {
       curtime = curtime + dt;
       double randvar = (double)randvec[counter];
       st = st * Math.Exp(drift + vsqrdt * randvar);
       counter = counter + 1;
    }
    callpayoffvec[i - 1] = Math.Max(st - _strike, 0.0);
    putpayoffvec[i - 1] = Math.Max(_strike - st, 0.0);}
    double callPrice = Math.Exp(-_riskFreeRate *
_maurity) * GetMean(callpayoffvec);
    double putPrice = Math.Exp(-_riskFreeRate *
_maurity) * GetMean(putpayoffvec);
    double[] returnValue = new double[] { callPrice, put-
Price };
   return returnValue;
}
//Returns an array of n normally distributed variables
using box muller transformation
public double[] NRandVars(double value)
{
   double[] randomArray = new double[(int)value + 1];
   double n2 = 0, v1 = 0, v2 = 0, tmp = 0, fac = 0;
   int counter = 0;
   n2 = Math.Floor((double)value / 2);
   counter = 0;
   Random random = new Random();
   for (long i = 1; i <= n2; i++)
   {
      do
      {
         v1 = 2 * random.NextDouble() - 1;
         v2 = 2 * random.NextDouble() - 1;
         tmp = v1 * v1 + v2 * v2;
      } while (!(tmp <= 1));
      fac = Math.Sqrt(-2 * Math.Log(tmp) / tmp);
      counter = counter + 1;
      randomArray[counter] = v1 * fac;
      counter = counter + 1;
      randomArray[counter] = v2 * fac;
   }
   if ((value > (n2 * 2)))
   {
      do
      {
         v1 = 2 * random.NextDouble() - 1;
         v2 = 2 * random.NextDouble() - 1;
         tmp = v1 * v1 + v2 * v2;
         } while (!(tmp <= 1));
         fac = Math.Sqrt(-2 * Math.Log(tmp) / tmp);
         counter = counter + 1;
         randomArray[counter] = v2 * fac;
   }
   return randomArray;
}
//Returns mean of an array
public double GetMean(double[] x)
{
   double tmpsum = 0;
   float n = x.GetUpperBound(0) - x.GetLowerBound(0)
+ 1;
   for (int i = x.GetLowerBound(0); i <=
x.GetUpperBound(0); i++)
   {
      tmpsum = tmpsum + x[i];
   }
   return tmpsum / n;
}
}
}
```

[C# code of the calculation class]

6 Evaluation Results

The presented framework has been evaluated on a test-bed consisting of 2, 4, 8 and 16 laptops. The laptops were connected through an Ethernet network using a traditional 100mbps switch. All simulation nodes had the same hardware specifications and operating system, namely processor: Intel Core i5 2.3 GHz; RAM: 4 GB; and Windows 7 (64 bit). One of the nodes was configured to play both the Manager and Executor with the client application also installed. The remaining nodes were only configured to act as executors.

To evaluate the efficiency of the grid-based Monte Carlo simulator, 10,000 runs (valuations) for European Options were simulated using different sets of variables. Each run was simulated 100,000 times to

obtain (nearly) accurate results. To further improve the accuracy, 10 time-steps for each run were used. This, on one hand, has notably improved the accuracy of the simulations, but on the other hand, it dramatically increased the total valuation time.

As mentioned earlier, to parallelize the simulation process of the implementation, the client application was configured to create a single job for each valuation task. Certainly, this configuration is not suitable for all Monte Carlo simulations; implementers should reasonably size and scope their grid jobs according to the complexity and length of the core valuation process. That is, creating long-running jobs that would take several minutes or hours, or very short jobs that would take only few milliseconds may negatively impact the overall performance of the deployed grid. In the former case, the jobs (runs) might run in a serial-like manner ignoring the advantage of having a grid of computers, whereas in the latter, a large number of network round trips would be incurred, and the Manager node might be overloaded with the incoming requests.

Before running the simulations on the deployed testbed, they were tested separately on a single machine. This enabled the grid-based model to be easily compared with the traditional (non-grid based) model. Fig. 6 illustrates how the grid-based simulation model enabled us to cut the total simulation time drastically. As shown, the simulation time was proportionally reduced with the number of installed computers. This speedup is directly associated with the fact that the grid computing model combines the power of the deployed computers to calculate different simulation paths/runs simultaneously.

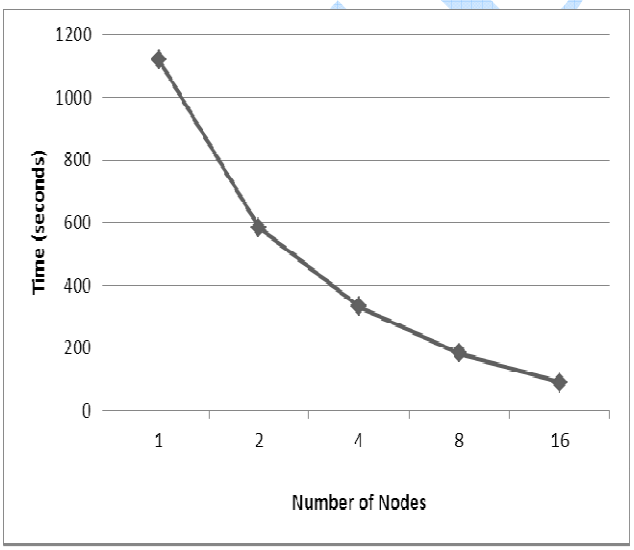

Fig. 6. Simulation time on Grid vs. Single Nodes

7 Conclusion

The utilization of enterprise grids allows businesses, organizations and educational institutions to effectively run Monte Carlo simulations without the burden of time, complexities or costs. This work presented a grid-enabled options' valuation framework based on the Monte Carlo method. The paper started with an introduction to the Monte Carlo method. A historical review for the related work was also presented to show how the proposed framework is different. The basic terms of financial options and grid computing were also given before discussing the proposed framework. Then, the implementation details including the architecture, used technologies and the C# code of the client application, EuropeanOptionPricing, EuropeanOptionPricingThread class were discussed.

Results show that the speed of the grid-enabled valuation for the European options using the Monte Carlo method increases proportionally with the number of connected workers. The readers might think that the presented simulations could run in zero or (near to zero) seconds if it is performed on a bigger and more powerful grid. In fact, this is not possible due to a number of factors including the underlying network topology and speed as well as the scheduling mechanism used by the grid middleware. However, implementers are still able to easily and efficiently obtain a supercomputer-like performance with only a fraction of the cost.

References

[And86] **H. L. Anderson** - *Metropolis: Monte Carlo and the MANIAC*. Los Alamos Science 14 (1986) 96–108

[ABM01] **A. Abdelkhalek, A. Bilas, A. Michaelides** - *Parallelization, Optimization, and Performance Analysis of Portfolio Choice Models*. Proceedings of the 30th International Conference on Parallel Processing (2001)

[Boy77] **P. P. Boyle** - *Options: A Monte Carlo approach*. Journal of Financial Economics. Vol. 4 (1977) 323–338

[BM58] **G. E. P. Box, M. E. Muller** - *A Note on the Generation of Random Normal Deviates*. The Annals of Mathematical Statistics. Vol. 29. No. 2 (1958) 610–611

[FK04] **I. Foster, C. Kesselman** - *The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure*. Elsevier (2004)

[FKT01] **I. Foster, C. Kesselman, S. Tuecke** - *The anatomy of the grid: Enabling scalable virtual organizations*, Int. J. High Performance Computing (2001) 200–222

[FS73] **B. Fischer, M. Scholes** - *The Pricing of Options and Corporate Liabilities*. Journal of Political Economy. Vol. 81. No. 3 (1973) 637-654

[Gla03] **P. Glasserman** - *Monte Carlo Methods in Financial Engineering (Stochastic Modeling and Applied Probability)*. Springer (2003)

[Hau06] **E. G. Haug** - *The Complete Guide to Option Pricing Formulas*. 2nd Edition, MsGraw-Hill (2006)

[Hul11] **J. C. Hull** - *Options, Futures, & Other Derivatives*. 8th Edition, Pearson College (2011)

[KP05] **C. Kolb, M. Pharr** - *Option pricing on the GPU*, GPU Gems 2. Chapter 45 (2005)

[L+05] **A. Luther, R. Buyya, R. Ranjan, S. Venugopal** - *Alchemi: A .NET-Based Enterprise Grid Computing System*. Proceedings of the 6th International Conference on Internet Computing (ICOMP'05), Las Vegas, USA (2005)

[M+07] **R. Moreno-Vozmediano, K. Nadiminti, S. Venugopal, A. B. Alonso-Conde, H. Gibbins, R. Buyya** - *Portfolio and investment risk analysis on global grids*, Journal of Computer and System Sciences, Elsevier (2007)

[Ros09] **S. M. Ross** - *Introduction to Probability Models*. 10th Edition, Elsevier (2009)

[TBG08] **X. Tian, K. Benkrid, X. Gu** - *High Performance Monte-Carlo Based Option Pricing on FPGAs*. IAENG Journal Engineering Letters, Special Issue on High Performance Reconfigurable Systems. Vol. 16. No. 3 (2008) 434-442

[***11a] http://www.investopedia.com/ (November, 2011)

[***11b] http://www.w3schools.com/browsers/browsers_os.asp (November, 2011)

[***11c] http://www.cloudbus.org/~alchemi/ (November, 2011)

[***11d] http://www.utilify.com/ (November, 2011)

[***11e] http://www.manjrasoft.com/products.html (November, 2011)

[***11f] http://www.digipede.net/ (November, 2011)

[***11g] http://sourceforge.net/projects/alchemi/ (November, 2011)