Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών,
Τηλεπικοινωνιών και Δικτύων

Διπλωματική Εργασία

ΧΡΗΜΑΤΟΟΙΚΟΝΟΜΙΚΑ ΠΑΡΑΛΛΗΛΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΜΟΝΤΕΛΑ

από

Άγγελος - Χρήστος Αναδιώτης

Επιβλέποντες:

1. Καθηγητής Ηλίας Χούστης
2. Επίκουρος Καθηγήτρια Παναγιώτα Τσομπανοπούλου

Βόλος, Οκτώβριος 2008

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών,
Τηλεπικοινωνιών και Δικτύων

Διπλωματική Εργασία

ΧΡΗΜΑΤΟΟΙΚΟΝΟΜΙΚΑ ΠΑΡΑΛΛΗΛΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΜΟΝΤΕΛΑ

από

Άγγελος - Χρήστος Αναδιώτης

Επιβλέποντες:

1. Καθηγητής Ηλίας Χούστης
2. Επίκουρος Καθηγήτρια Παναγιώτα Τσομπανοπούλου

Βόλος, Οκτώβριος 2008

Contents

Chapter 1

Πρόλογος στην Ελληνική Γλώσσα

Μέσα σε ένα έντονα ανταγωνιστικό περιβάλλον, όπως είναι αυτό των χρηματοοικονομικών, όπου δημιουργούνται νέα, πιο πολύπλοκα, προϊόντα, η ανάγκη για επιτάχυνση και ακρίβεια στους υπολογισμούς γίνεται ολοένα και πιο επιτακτική. Για να καλυφθούν αυτές οι νέες ανάγκες, με τα ήδη υπάρχοντα μαθηματικά - υπολογιστικά μοντέλα, προχωράμε σε παραλληλοποίηση των υπολογισμών. Με αυτόν τον τρόπο επιτυγχάνουμε την επίταχυνσή τους, ένα ποσοστό της οποίας θα μπορούσε να θυσιαστεί για χάρη της βελτίωσης της ακρίβειας τους.

Η εν λόγω διπλωματική εργασία αποτελεί μια προσπάθεια αποτύπωσης των βασικών υπολογιστικών μοντέλων που χρησιμοποιούνται στα χρηματοοικονομικά, ενώ ταυτόχρονα επιχειρήθηκε και μια διαφορετική προσέγγιση στην παραλληλοποίησή τους. Μέσα σε αυτό το πνεύμα, από τη μία πλευρά παρουσιάζονται τρεις μέθοδοι αποτίμησης χρηματοοικονομικών παραγώγων σε θεωρητικό επίπεδο, δίνοντας το απαραίτητο μαθηματικό υπόβαθρο, σε πρακτικό επίπεδο, δίνοντας αλγορίθμους αποτίμησης που αποτυπώνουν τη θεωρία και, τέλος, σε ένα πιο προχωρημένο επίπεδο, δίνοντας τους αντίστοιχους παράλληλους υπολογιστικούς αλγορίθμους. Τέλος, η διπλωματική αυτή εργασία συνοδεύεται και από τον πηγαίο κώδικα που υλοποιεί τους παράλληλους υπολογισμούς για Ευρωπαϊκά options σε συνδυασμό με ένα Web Interface που καθιστά τα εργαλεία αυτά πιο εύχρηστα.

Σε αυτό το σημείο, να σημειωθεί ότι η γλώσσα γραφής που χρησιμοποιήθηκε για την παρουσίαση των μοντέλων είναι η αγγλική. Στο Παράρτημα 2 παρατίθεται το εν λόγω κείμενο στην αγγλική γλώσσα. Στη συνέχεια, θα δώσουμε την περιγραφή της διάρθρωσης του κειμένου.

Η διπλωματική αυτή εργασία αποτελείται από ένα εισαγωγικό κεφάλαιο στην ελληνική και ένα παράρτημα, όπου παρατίθεται το υπόλοιπο κομμάτι στην αγγλική. Το κομμάτι της που είναι στην αγγλική, αποτελείται από τέσσερα κεφάλαια, εκ των οποίων το πρώτο είναι εισαγωγικό, και τρία παραρτήματα. Ακολουθεί μια σύντομη παρουσίαση αυτών που παρουσιάζονται σε αυτό το κομμάτι, τα οποία και αποτελούν το βασικό πυρήνα της εργασίας.

Στην Εισαγωγή - Κεφάλαιο 1, γίνεται μια παρουσίαση των βασικών χαρακτηριστικών του περιβάλλοντος των χρηματοοικονομικών παραγώγων με τα οποία θα ασχοληθούμε παρακάτω.

Στις Μεθόδους Monte Carlo - Κεφάλαιο 2, δίνονται οι βασικές αρχές που διέπουν αυτές τις μεθόδους, τις ανάγκες και τα προβλήματα που ανακύπτουν καθώς και λύσεις σε αυτά μέσα από τη βιβλιογραφία. Τέλος, δίνονται αλγόριθμοι υπολογισμού Ευρωπαϊκών options με χρήση σειριακών και παράλληλων μεθόδων Monte Carlo.

Στις Μερικές Διαφορικές Εξισώσεις - Κεφάλαιο 3, παρουσιάζεται η πιο διαδεδομένη μέθοδος επίλυσής τους και γίνεται επίδειξή της μέσα από ένα παράδειγμα, από το οποίο μπορούν να προκύψουν λύσεις που αντιστοιχίζονται σε προβλήματα στα χρηματοοικονομικά. Τέλος, δίνονται παράλληλοι αλγόριθμοι για την αποτίμηση Ευρωπαϊκών options με χρήση παράλληλων μεθόδων επιλύοντας μια Μερική Διαφορική Εξίσωση.

Στα Διωνυμικά Δέντρα - Κεφάλαιο 4, παρουσιάζεται ακόμα μια μέθοδος αποτίμησης παραγώγων. Αφού γίνει μια αρχική περιγραφή της μεθόδου, δίνονται ο σειριακός και ο παράλληλος αλγόριθμος αποτίμησης Ευρωπαϊκών options .

Στα Πειράματα - Κεφάλαιο 5, δίνονται οι γραφικές παραστάσεις από μετρήσεις που έγιναν σε προσομοιώσεις του παράλληλου πηγαίου κώδικα. Έχουν προκύψει από πειραματικά αποτελέσματα και μπορούν να δώσουν μια καλή εικόνα του γιατί να επιλέξουμε τους παράλληλους υπολογισμούς.

Στα Συμπεράσματα-Μελλοντικό Έργο - Κεφάλαιο 6, παρουσιάζονται κάποια γενικά συμπεράσματα που έχουν προκύψει μέσα από αυτή τη διπλωματική εργασία και την αντίστοιχη έρευνα που έγινε για να εκπονηθεί. Επίσης, αναφέρονται κάποιες κατευθηντήριες σχετικά με την επέκταση και τη συνέχεια αυτής της εργασίας.

Στο Παράρτημα Α δίνονται κάποιες πληροφορίες για την ουδετερότητα ρίσκου, την οποία πολλές φορές θεωρούμε δεδομένη και είναι σημαντικό να γνωρίζουμε

γιατί το θεωρούμε αυτό και σε ποιές περιπτώσεις μπορεί να επιτραπεί αυτή η πιο "απλοϊκή' προσέγγιση στα μοντέλα μας.

Στο Παράρτημα Β γίνεται μια αναφορά στα Πραγματικά options (*Real options*) . Δίνεται ο ορισμός τους και κάποια παραδείγματα αυτών των παραγώγων καθώς και το πώς μπορούν να επηρεάσουν την τιμή της μετοχής μιας επιχείρησης.

Στο Παράρτημα Γ δίνονται κάποιες τεχνικές λεπτομέρειες που έχουν να κάνουν με τη λειτουργία του λογισμικού που δίνεται μαζί με αυτή την εργασία.

Chapter 2

Παράρτημα

Ακολουθεί το κείμενο της Διπλωματικής Εργασίας γραμμένο στην αγγλική γλώσσα.

University of Thessaly
School of Engineering
Department of Computer and Communication Engineering

Diploma Thesis

Parallel Computational Models in Finance

by

Angelos - Christos Anadiotis

Supervisors:

1. Professor Elias Houstis
2. Assistant Professor Panagiota Tsompanopoulou

Volos, October 2008

To my family

Contents

List of Figures

List of Algorithms

Acknowledgements

First of all, I would like to thank Professor Elias N. Houstis and Assistant Professor Panagiota Tsompanopoulou for their trust and support for this thesis, for my studies, for my future. Then my family for their being so supportive during all these years of my studies at University. My closest friends Nikos Papavasiliou and Antonis Gogakis for their being patient with me. PhD candidate Dimitris Syrivellis and technical support member Thanasis Fevgas for their advice and assistance when I was dealing with problems for the simulations. A very special person for keeping me calm and peaceful when everything seemed so hard.

Chapter 1

Introduction

1.1 Preface

Finance is a branch of economics concerned with resource allocation as well as resource management, acquisition and investment[7]. It is one of the fastest developing areas in the modern banking and corporate world. What we are dealing with in the context of this Thesis, is computational models for financial markets, the assets that are traded in them and financial derivative products. There are many kinds of financial markets; the most important are:

- **Stock Markets**
- **Bond Markets**
- **Currency Markets**
- **Commodity Markets**
- **Options** and **Futures Markets**

We have, then, a collection of markets on which assets of various kinds are bought and sold. In the beginning it was just a buy/sell trading of assets in any form (i.e. stocks for stock markets, bonds for bond markets etc.). However markets have become more sophisticated and investors need a greater range of opportunities to tailor their dealings to their investment needs. This range is getting greater with several products known as *financial derivatives*, *derivative securities*, *derivative products*, *contingent claims* or just *derivatives*.

1.2 Options

Option is a contract that gives its owner the right -but not the obligation- to buy or sell a prescribed asset, currency and, in general, any commodity amount in a prescribed price and at a prescribed time in the future (or during this prescribed time in the future).

When an investor buys the right to buy (or sell) a title, then the owner of the title, known as the *writer*, is obligated to sell (or buy) this title when the option holder asks for it. The option holder is known as the *buyer of the option* or he is said to be *going long on the option.* The other party is known as the *seller of the option* or he is said to be *going short on the option.* The option buyer pays the seller an amount which is called *premium*; premium is just a small part of the commodity value. The prescribed price of buying or selling this commodity is called *exercise price* or *strike price.*

1.2.1 Options Types

Depending on exercise time

If an option can be exercised only at its prescribed expiration time, then it is said to be a *European Option.* On the other hand, if the option holder has the right to exercise the option anytime until the expiration time, then the option is said to be an *American Option.*

An American option can be considered as a batch or European options with the same exercise price and with each one to begin when the previous one has expired. In the context of this Thesis, we are dealing only with European options, since we can present all the basic option properties can be illustrated through them.

Depending on the transaction

If the option gives the holder the right to buy or sell the underlying commodity, then it is called *call option* or *put option* respectively.

Call Options

A **call option** is a financial contract between two parties, the buyer and the seller of this type of option. Often it is simply labelled a "call". The option holder has the right to buy an agreed quantity of a particular commodity or financial instrument (the underlying instrument) from the seller of the option at a certain time (the expiration date) for a certain price (strike price). The writer

is obligated to sell the commodity or the financial instrument, should the buyer so decide. The buyer pays the premium for this right.

If the commodity price S_T is greater than the exercise price K of the option, then the call option holder is using his right by buying this commodity at the specified -strike- price and right after that he is selling the commodity at S_T so that he wins immediately $S_T - K$. His final profit is $S_T - K - [premium]$.

On the other hand, if the commodity price is less than K, then the call option holder is not using his right and loses the premium.

The writer of the call option wins the premium and, in the first case he loses $S_T - K$ while in the second case he loses nothing and he has the premium as his profit.

Put Options

A **put option**, or simply a "put", is a financial contract between two parties, the seller and the buyer of the option. The put option allows its buyer the right to sell a commodity or a financial instrument to the writer of the option at a certain time (expiration date) for a certain price (strike price). The writer is obligated to purchase the underlying asset at that strike price, if the holder exercises the option.

A put holder is using his right when the commodity value is less than the exercise price K at the option expiration date. In this case, the profit is $K - S_T$, where S_T is the underlying value. Putting it all together, the final profit for the holder is $K - S_T - [premium]$. If the holder wants to get this profit, all he has to do is to buy the commodity at its price S_T and sell it at the exercise price K.

On the other hand, if the asset price S_T is greater than the exercise price K, the put option holder is not using his right and loses the premium.

A put writer, gets the premium from the buyer and, in the first case he loses $K - S_T$ while in the second case he loses nothing and he has the premium as his profit.

Depending on the relation between the exercise price and the asset price

Depending on the relation between the exercise price of the option and the asset price, an option can be

- **at-the-money**, if the exercise price equals the underlying asset price,
- **in-the-money**, if the asset price is greater than the exercise price of the call option (for a put option the asset price must be less than the exercise price),
- **out-of-the-money**, if the exercise price is greater than the asset price for a call option (for a put option the exercise price must be less than the asset price).

It is obvious that *in-the-money* options are of greater value than *at-the-money* options and *at-the-money* options are of greater money than *out-of-the-money* options.

For a more detailed approach on financial derivatives please refer to [19].

1.3 Computational Models in Finance

In this Thesis we examine the modelling of financial derivative products from the applied mathematics viewpoint, from modelling through analysis to elementary computation. We are using the three most common computational methods applied in finance:

- **Monte Carlo** at Chapter 2,
- **Partial Differential Equations** at Chapter 3,
- **Binomial Trees** at Chapter 4.

Each one of these methods is illustrated in the context of this document. We use European Options to present our examples and show how these methods are applied in the real world. Based on the financial theory and these computational methods, we form real-time financial problems on derivative securities and we are solving them providing numerical solutions.

For the Monte Carlo method, we are using the asset price random walk and we simulate different possible paths.

For the Partial Differential Equations method, we are using the **Black-Scholes** model for European Options. Then, we are solving this stochastic partial differential of parabolic type using the finite-difference scheme and more specifically the following schemes:

- **Explicit Finite-Differences** illustrated at 3.2.1.1,

- **Implicit Finite-Differences** illustrated at 3.2.1.2 and
- **Crank-Nicolson** illustrated at 3.2.1.3.

For the binomial trees method, we are using the **Cox, Ross** and **Rubinstein** model.

However, this is not all. We are parallelizing these methods above, so that we can speed up the computations for two reasons: *time* and *accuracy*. The first one is obvious. The second is a result of the first; meaning that in the same time, we can run more loops of the simulations, which can improve the accuracy of our numerical results.

1.4 Thesis Structure

The rest of the Thesis is structured as follows.

Chapter 2 is a reference in Monte Carlo methods in option pricing. First, the methods semantics and demands are demonstrated so that they are better understood and then they are used in option pricing.

Chapter 3 is dedicated in Partial Differential Equations. The Finite-Difference method is demonstrated along with three schemes that can be used in solving PDEs. Then, these methods are used in option pricing.

Chapter 4 demonstrates Binomial Trees in option pricing. Starting from its birth, we show how an asset random walk can be modelled by a binomial tree with certain parameters.

Chapter 5 contains a set of figures and results from experiments ran using the source code that is a part of this Thesis.

Chapter 6 gives the reader the conclusions that we have come to in the context of this Thesis and is a plan for what could come next.

In appendices the reader can find additional information on certain topics. More specifically:

Appendix A gives information on Risk Neutrality.

Appendix B is a first approach to Real Options.

Appendix C gives a short description about the implementation (code, GUI) and where to find it.

Chapter 2

Monte Carlo Methods

2.1 Introduction

A Monte Carlo method is a computational algorithm that relies on repeated random sampling to compute its results. Monte Carlo methods are widely applicable in the field of mathematics and physics. They were invented by physics researchers during the World War II, but due to their high computational requirements, they were not used massively, until the first electronic computers were built. Since then, as the computers performance raise, Monte Carlo methods become more and more popular to researchers in various fields.

In finance, Monte Carlo methods are used to analyse and value basic financial models through to complex instruments, portfolios and investments by simulating the sources of uncertainty affecting their value. After simulating a large number of random walks which provide different values, Monte Carlo methods consider the final value as the average of all values that came out of the simulation process.

As the problems become more complex by adding more sources of uncertainty, Monte Carlo methods gain the upper hand among other ones. But as the simulations become more demanding, the need to speed up the calculations is even more essential. The parallelism of Monte Carlo methods is a solution to that problem.

2.2 Monte Carlo Algorithm

When we are trying to value a derivative security (such as an option), we need to know the prices of the underlying securities. These prices are often modelled as continuous-time stochastic processes. So, a Monte Carlo algorithm has to

simulate as many different paths of the underlying asset as possible and, through the law of large numbers, estimate the expected value of the option, as the mean of the values that resulted from every different path. This results to Algorithm 1 as shown in [16].

Algorithm 1 Monte Carlo

1: **for** $j = 1$ to N **do**
2: Simulate sample paths of the underlying variables (asset prices, interest rates, etc.) using the risk neutral measure over the time frame of the option. For each simulated path, evaluate the discounted cash flows of the derivative C_j.
3: **end for**
4: Average the discounted cash flows over the sample paths $\hat{C} = \frac{1}{N}\sum_{j=1}^{N} C_j$
5: Compute the standard deviation $\hat{\sigma}_{\hat{C}} = \sqrt{\frac{1}{(N-1)}\sum_{j=1}^{N}(C_j - \hat{C})^2}$

2.3 Serial Random Number Generators

In computer systems, a random number generator is an algorithm designed to generate a sequence of numbers that lack any pattern. Even though, there are some statistical tests for randomness, intended to ensure that the numbers produced do not have easily discernible patterns, computer-based systems often fail to generate random numbers. The most commonly used methods [16] are the following.

2.3.1 Linear Congruential Generator (LCG)

One of the most commonly used random number generator is the linear congruential generator. It is based on the recurrence

$$y_n = (\alpha y_{n-1} + c) \mod m$$

where $m > 0$ is the modulus, $\alpha > 0$ is the multiplier and c the additive constant. It is usually denoted $LCG(m, \alpha, c, y_0)$, where y_0 is the initial value for the recurrence. Due to the modulus, the numbers generated by this generator have a maximum period of m.

2.3.2 Multiplicative Linear Congruential Generator (MLCG)

This generator is a result of the previous one, if $c = 0$. In this case, the recurrence becomes

$$y_n = (\alpha y_{n-1}) \mod m$$

If we choose the appropriate parameters, the multiplicative linear congruential generator can produce a sequence of numbers of maximal period.

2.3.3 Minimal Standard (MINSTD)

For 32-bit machines, the choice $LCG(2^{31}-1, 16807, 0, 1)$, also known as MINSTD for minimal standard, is a popular one.

2.3.4 Multiple-Recursive Generator

This method proposed by L'Ecuyer extends MRG by adding k terms in the recurrence

$$y_n = (\alpha_1 y_{n-1} + \alpha_2 y_{n-2} + \ldots + \alpha_k y_{n-k}) \mod m$$

where $(\alpha_i)_{i=1}^{k}$ are integers in the interval $[-(m-1), (m+1)]$. The period and the randomness of the numbers produced by this generator are generally much improved compared with an MRG at the cost of an increase of computation time.

2.4 Parallel Random Number Generators

2.4.1 Leapfrog

The leapfrog method distributes the numbers of a serial random number geneartor in a cyclic fashion to each processor. If we denote by $(x_i)_{i=0,1,2,\ldots}$ the original sequence and L the lag, then processor p gets the following subsequence:

$$\tilde{x}_i = x_{iL+p} \qquad with \qquad p = 0, 1, 2, \ldots, P \leq L-1$$

If the original sequence is

$$x_0, x_1, \ldots, x_{L-1}, x_L, x_{L+1}, \ldots, x_{2L-1}, x_{2L}, x_{2L+1}, \ldots$$

then the subsequence obtained by processor 0 is

$$\boxed{x_0}, x_1, \ldots, x_{L-1}, \boxed{x_L}, x_{L+1}, \ldots, x_{2L-1}, \boxed{x_{2L}}, x_{2L+1}, \ldots$$

There are two problems with this method:

- Long-range correlations embedded in the random number generator can become short-range correlations in the new sequence and destroy the quality of the parallel random number generator.
- Not scalable scheme, since, when the number of processors P increases, the length of the sequence $(\tilde{x}_i)_{i=0,1,2,\ldots}$ decreases.

2.4.2 Sequence Splitting

The original sequence is split into blocks and distributed to each processor. Let us denote the period of the generator by ρ, the number of processors by P and the block length by $L = \lfloor \rho/P \rfloor$, we have

$$\hat{x}_i = x_{pL+i} \qquad \text{with} \qquad p = 0, 1, 2, \ldots, P$$

Then, the original sequence

$$x_0, x_1, \ldots, x_{L-1}, x_L, x_{L+1}, \ldots, x_{2L-1}, x_{2L}, x_{2L+1}, \cdots$$

is distributed as follows to processors $0, 1, 2, \ldots$

$$\boxed{x_0, x_1, \ldots, x_{L-1}} \boxed{x_L, x_{L+1}, \ldots, x_{2L-1}} \boxed{x_{2L}, x_{2L+1}, \ldots, x_{3L-1}} \cdots$$

There two problems with this method:

- Long-range correlations can be emphasized and become inter-processor correlations. We know that the sequences produced will not overlap, but we cannot be sure that they will not show some correlation.
- Not scalable scheme, since, when the number of processors P increases, the length of the sequence $(\tilde{x}_i)_{i=0,1,2,\ldots}$ decreases.

2.4.3 Parameterization

This method is based on parametrization of each stream of numbers. This can be done in two ways:

- In certain generators, the seed value provides a natural way of dividing the sequence of a random number generator into independent cycles.
- The function that outputs the next value in the sequence can be parametrized to give a different stream for a different value.

2.5 Variance Reduction Techniques

There are several techniques used to reduce variance. Here, we will examine the four most common ones and present them, as illustrated in [17, 1]

2.5.1 Variance Reduction and Efficiency Improvement

The reduction of variance is obviously desirable for many reasons. In this Section we will examine it from the perspective of improving the computational efficiency.

Suppose we want to compute the value of a derivative security -let us denote it by θ. In order to do so, we use Monte Carlo methods to generate an i.i.d. (independent and identically distributed) sequence $\{\hat{\theta}_i, i = 1, 2, \ldots\}$, where each $\hat{\theta}_i$ has expected value θ and variance σ^2. A natural estimator of θ based on n replications is then the sample mean

$$\frac{1}{n}\sum_{i=1}^{n}\hat{\theta}_i.$$

By the central limit theorem, for a large n, this sample mean is approximately normally distributed with mean θ, variance σ^2/n and error proportional to σ/n. Thus, decreasing the variance σ^2 by 10, and leaving all the rest unchanged, does as much for error reduction as increasing the number of samples by a factor of 100.

In case we have to choose between two Monte Carlo sequences, we should keep in mind that the variance is not the only factor that should affect our final choice. For example, let us suppose that we have two Monte Carlo sequences to value a derivative security θ. Let us denote these two sequences by $\{\hat{\theta}_i^{(1)}, i = 1, 2, \ldots\}$ and $\{\hat{\theta}_i^{(2)}, i = 1, 2, \ldots\}$. Suppose that both are unbiased, so that $E[\hat{\theta}_i^{(1)}] = E[\hat{\theta}_i^{(2)}] = \theta$, but $\sigma_1 < \sigma_2$. Based on our previous observations, we should conclude that the first sequence is the most proper one to choose as it gives a more precise estimate of θ than the second one. In this case, we should check the case that the first sequence is more computationally expensive than the second one, and thus it has smaller variance. So, we need to find a formula that takes care of this case too.

In order to derive this formula, we work as follows. Let us denote a constant $b_j, j = 1, 2$, as the work required to generate one replication of $\hat{\theta}^{(j)}$. Assuming computations time t, the number of replications of $\hat{\theta}^{(j)}$ that can be generated will be $\lfloor t/b_j \rfloor$ (for simplicity we consider the ratios t/b_j to be integers). Thus, the two estimators are:

$$\frac{b_1}{t}\sum_{t=1}^{t/b_1}\hat{\theta}_i^{(1)} \quad \textit{and} \quad \frac{b_2}{t}\sum_{t=1}^{t/b_2}\hat{\theta}_i^{(2)}$$

For large t, these estimators are approximately normally distributed with mean θ and standard deviations

$$\sigma_1\sqrt{\frac{b_1}{t}} \quad \textit{and} \quad \sigma_2\sqrt{\frac{b_2}{t}}$$

Therefore, for large t the first estimator should be preferred over the second in case that

$$\sigma_1^2 b_1 < \sigma_2^2 b_2 \tag{2.1}$$

Equation (2.1) gives us a trade-off for estimator variance and computational efficiency. Based on our conclusions, it is reasonable to use the product of the variance with the computational work per run as a measure of efficiency. Using efficiency as a basis for comparison, the lower variance estimator should be preferred only if the variance ratio σ_1^2/σ_2^2 is smaller than the work ratio b_2/b_1 and vice versa.

2.5.2 Antithetic Variates

The antithetic variates technique is one of the simplest and most commonly used techniques in financial pricing problems. We are going to use as an example the pricing of a european call option on a no-dividend stock to illustrate the method.

Based on the Black-Scholes model, the stock price follows a log-normal diffusion. Independent replications of the terminal stock price under the risk-neutral measure can be generated from the formula:

$$S_T^{(i)} = S_0 e^{(r-(1/2)\sigma^2)T+\sigma\sqrt{T}Z_i}, \quad i = 1, \ldots, n, \tag{2.2}$$

where S_0 is the current stock price, r is the riskless interest rate, σ is the stock's volatility, T is the option's maturity and the $\{Z_i\}$ are independent samples from the standard normal distribution. Based on n replications and assuming an exercise price K, an unbiased price of the option is given by

$$\hat{C} = \frac{1}{n}\sum_{i=1}^{n} C_i \equiv \frac{1}{n}\sum_{i=1}^{n} e^{-rT}\max\{0, S_T^{(i)} - K\} \tag{2.3}$$

Antithetic variates technique is based on the fact that, if $\{Z_i\}$ has a standard normal distribution, then so does $\{-Z_i\}$. By replacing Z_i with $-Z_i$ in equation (2.2), we obtain the stock price $\tilde{S}_T^{(i)}$ and then the price of the option:

$$\tilde{C}_i = e^{-rT}\max\{0, \tilde{S}_T^{(i)} - K\} \tag{2.4}$$

Finally, the price of the call option will be

$$\tilde{C}_{AV} = \frac{1}{n}\sum_{i=1}^{n} \frac{C_i + \tilde{C}_i}{2} \tag{2.5}$$

The antithetic pairs $\{(Z_i, -Z_i)\}$ are more regularly distributed than any other $2n$ totally independent samples. The reason for this is that the sample mean of the antithetic pairs is 0; which is very unlikely for any other set of numbers. Thus, we may hope that, if the inputs are made more regular, then we will have more regular outputs as well.

For the shake of formality, we are going to compare efficiencies. Since C_i and $\tilde{C}_i$ have the same variance, we have that

$$Var\left[\frac{C_i + \tilde{C}_i}{2}\right] = \frac{1}{2}(Var[C_i] + Cov[C_i, \tilde{C}_i]) \tag{2.6}$$

In order to prove our claim, that is $Var[\hat{C}_{AV}] \leq Var[\hat{C}]$, it has to be $Cov[C_i, \tilde{C}_i] \leq Var[C_i]$. Since computing $\hat{C}_{AV}$ needs twice as many replications as $\hat{C}$ we need to check the computational requirements. Considering that generating Z_i takes a negligible fraction of the work per replication, then the total work to generate $\hat{C}_{AV}$ is roughly double the work to generate $\hat{C}$. So, in order for this technique to be useful, it must be

$$2Var[\hat{C}_{AV}] \leq Var[\hat{C}]$$

By replacing the above equation in (2.6), we get the final requirement, that

$$Cov[C_i, \tilde{C}_i] \leq 0$$

In order to prove the last case, we work as follows. We consider a function ϕ such that $C_i = \phi(Z_i)$. This means that ϕ is the composition of the mappings from Z_i to the stock price and from the stock price to the discounted option payoff. As being a composition of two monotones, ϕ is also a monotone, so by a standard inequality

$$E[\phi(Z_i), \phi(-Z_i)] \leq E[\phi(Z_i)]E[\phi(-Z_i)] \tag{2.7}$$

Thus,

$$Cov[C_i, \tilde{C}_i] \equiv E[\phi(Z_i), \phi(-Z_i)] - E[\phi(Z_i)]E[\phi(-Z_i)] \leq 0$$

and we may conclude that this technique improves efficiency.

When we are computing the confidence intervals with antithetic variables, the standard error must be estimated using the sample standard deviation of the n averaged pairs $(C_i + \tilde{C}_i)/2$ rather than the $2n$ individual observations $C_1, \tilde{C}_1, \ldots, C_n, \tilde{C}_n$, since only the first are independent. In this case, the use of a variance reduction technique affects the estimation of the standard error and requires some "batching" of observations to deal with dependence.

2.5.3 Control Variates

This method is one of the most popular methods for variance reduction, because it is both effective and easy to use. It can make use of other known values to evaluate an unknown variable.

The most straightforward implementation of control variates replaces the evaluation of an unknown expectation with the evaluation of the difference between the unknown quantity and another expectation whose value is known. We will see its use through an example. Let P_A be an option price whose payoff depends on arithmetic average and P_G an option price whose payoff depends on geometric average. In most cases, arithmetic average is of much more use than geometric average; but instead of geometric average, arithmetic average cannot be evaluated in a closed form. In such a case, we can use P_G to compute P_A with the control variate method.

Let's say that $\hat{P}_A$ and $\hat{P}_G$ are the discounted option payoffs for a single simulated path of the underlying asset and $P_A = E[\hat{P}_A]$ and $P_G = E[\hat{P}_G]$. Then,

$$P_A = P_G + E[\hat{P}_A - \hat{P}_G]$$

So, now we have expressed P_A as the sum of P_G -which is known- with the expected difference between $\hat{P}_A$ and $\hat{P}_G$. An unbiased estimator is thus provided by

$$\hat{P}_A^{cv} = \hat{P}_A + (P_G - \hat{P}_G) \tag{2.8}$$

Using this scheme, we can adjust the estimator $\hat{P}_A$ according to the difference between the known value P_G and the observed value $\hat{P}_G$. The known error $(P_G - \hat{P}_G)$ is used as a control in the estimation of P_A.

The variance of $\hat{P}_A^{cv}$ is

$$Var[\hat{P}_A^{cv}] = Var[\hat{P}_A] + Var[\hat{P}_G] - 2Cov[\hat{P}_A, \hat{P}_G]$$

So, this method is considered to be effective, if the covariance between $\hat{P}_A$ and $\hat{P}_G$ is large enough. There are numerical results that indicate that this is indeed the case.

We can improve equation (2.8) by considering the family of unbiased estimators:

$$\hat{P}_A^{\beta} = \hat{P}_A + \beta(P_G - \hat{P}_G) \tag{2.9}$$

which are parametrized by the scalar β. In this case we have:

$$Var[\hat{P}_A^{\beta}] = Var[\hat{P}_A] + \beta^2 Var[\hat{P}_G] - 2\beta Cov[\hat{P}_A, \hat{P}_G]$$

So, the β which is minimizing the variance is

$$\beta^* = \frac{Cov[\hat{P}_A, \hat{P}_G]}{Var[\hat{P}_G]} \tag{2.10}$$

An estimator based on β^* is guaranteed not to increase variance and will result in a strict decrease in variance, as long as $\hat{P}_A$ and $\hat{P}_G$ are not uncorrelated.

But in practice, we rarely know β^* because we rarely know $Cov[\hat{P}_A, \hat{P}_G]$. However, given n independent replications $\{(P_{A_i}, P_{G_i}), i = 1, \ldots, n\}$ of the pairs $(\hat{P}_A, \hat{P}_G)$ we can estimate β^* via regression. At this point we face a choice. Using all n replications to compute an estimate $\hat{\beta}$ of β^* introduces a bias in the estimator

$$\frac{1}{n}\sum_{i=1}^{n} P_{A_i} + \hat{\beta}\left(P_G - \frac{1}{n}\sum_{i=1}^{n} P_{G_i}\right)$$

and its estimated standard error because of the dependence between $\hat{\beta}$ and the P_{G_i}. Reserving n_1 replications for the estimation of β^* and the remaining $n - n_1$ replications for the sample mean of the P_{G_i} (typically when $n_1 \ll n$) eliminates the bias but may deteriorate the estimate of β^*. Neither issue significantly limits the applicability of the method, because the possible bias vanishes as n increases and because the estimator of β^* need not be very precise to achieve a reduction in variance.

The advantage of working with (2.9) over (2.8) becomes even more pronounced when further controls are introduced. For example, when the asset price is simulated under risk-neutral probabilities, the present value $e^{-rT}E[S_T]$ of the terminal price must equal the current price S_0. We can, therefore, form the estimator

$$\hat{P}_A + \beta_1(P_G - \hat{P}_G) + \beta_2(S_0 - e^{-rT}S_T)$$

The variance-optimising coefficients (β_1^*, β_2^*) are easily found by multiple regression. This optimisation step seems particularly crucial in this case; for, whereas one might guess that β_1^* is close to 1, it seems unlikely that β_2^* would be. Optimising over the βs also allows us to exploit controls that are negatively correlated with the option payoff.

2.5.4 Moment Matching Methods

Here, we introduce a variance reduction technique, called *quadratic resampling.* This technique is based on *moment matching.* It will be illustrated through an example of estimating the European call option price on a single asset and then generalise.

As we already know, the asset prices are generated from the formula:

$$S_T(i) = S_0 e^{(r-(1/2)\sigma^2)T + \sigma Z_i\sqrt{T}}$$

where $Z_i, i = 1, \ldots, n$ denote independent standard normal random variables that drive the simulation. The sample moment of these n Z's will not exactly match those of the standard normal. The idea of moment matching is to transform the

Z's to match a finite number of the moments of the underlying population. The first of the standard normal can be matched by defining

$$\tilde{Z}_I = Z_i - \tilde{Z}, \quad i = 1, \ldots, n \tag{2.11}$$

where $\tilde{Z} = \sum_{i=1}^{n} \frac{Z_i}{n}$ is the sample mean of the Z's. Note that the $\tilde{Z}_i$'s are normally distributed if the Z_i's are normal.

So, if we go back to the asset price generator formula, we can see that a moment matching estimator of a call option price is the average of the n values $\tilde{C}_i = e^{-rT} \max(\tilde{S}_T(i) - K, 0)$.

But there is one drawback about this method. As we already know, in the standard Monte Carlo method, confidence intervals for the true value of C could be estimated by the sample mean and variance of the estimator -this cannot be done here, since the n values of $\tilde{C}$ are not independent. So, we need to apply moment matching to independent batches of runs and estimate the standard error from the batch means. This reduces the efficacy of the method compared with matching moments across all runs.

The method, as it was illustrated until now, considers the first moment of a distribution with zero mean. If this is not the case, then we use the equation

$$\hat{Z}_i = Z_i - \hat{Z} - \mu_Z, \tag{2.12}$$

where μ_Z is the population mean.

If we would like to match two moments of a distribution, then the appropriate transformation would be

$$\hat{Z}_i = \left(Z_i - \hat{Z}\right) \frac{\sigma_Z}{s_Z} + \mu_Z, \quad i = 1, \ldots, n, \tag{2.13}$$

where s_Z is the standard deviation of the Z_i's and σ_Z is the population standard deviation. For a standard normal, $\mu_Z = 0$ and $\sigma_Z = 1$. An estimator for a call option price is the average of the n values of $\hat{C}_i$.

Using the transformation (2.13), the Z_i's are normal. Hence, the corresponding $\hat{C}_i$ are biased estimators of the true option value. For most financial problems of practical interest, this bias is likely to be small. However, the bias can be arbitrarily large in extreme circumstances (even when only the first moment of the distribution is matched). The dependence and bias in the moment matching method makes it difficult to quantify the improvement in general analytical terms.

The moment matching method can be applied to matching higher order moments as well. In addition to different methods for transforming random outcomes to match specified moments, additional points could be added as another way to match moments.

Whenever a moment is known, it can be used as a control rather than for moment matching.

2.6 Option Pricing Using Monte Carlo Methods

Monte Carlo methods are, perhaps, the most popular ones in option pricing. Even though it is considered to be rather slow, since too many computations are needed for a trustworthy result, they are widely used mostly because of their simplicity. But Monte Carlo methods are easily parallelizable as well; this way we can reduce the time needed for computations, as shown in the experimental results. In the following Section, we will examine Monte Carlo and Parallel Monte Carlo methods in European Options pricing.

2.6.1 European Options

Consider a European Call Option, whose value at expiry is $E[\max(S_T - K, 0)]$, where S_T is the underlying security price at expiry. Our goal is to price that option by evaluating the discounted present value of S_T. In order to do so, we should first use a model to simulate the asset price random walk; let it be a Wiener stochastic process. Then, we are going to evaluate the option price at expiry. Every loop in Monte Carlo simulation, gives us another asset price at time T. All we have to do, is to gather those values and get their mean; this way we have a good approximation of the underlying security price at expiry.

The model that describes the underlying security random walk is[13, 18, 21]:

$$S_{t+1} = S_t e^{(r - \frac{\sigma^2}{2})\delta t + \sigma\epsilon\sqrt{\delta t}}, \tag{2.14}$$

where ϵ is a random sample from a standard normal distribution, with zero mean and deviation one.

After calculating the asset price random walk, we reach to its price at expiry. At this point we can evaluate the call option as

$$C = e^{-rT} \max(S_T - K, 0)$$

At this point, we have finished one step of the simulation. After doing the same thing for many more times, we gather the results and then compute their mean,

which will be the option price. Then, we can find the standard deviation of the expected value and based on it, we can calculate the probability for the call option price to belong in a certain space.

In case we have more than one stochastic variables, then all we have to do, is to follow the same procedure described above; only that this time, we will have to get the random walks for all of the stochastic variables. In case the interest rate r is a stochastic variable, then we have to find the random walk of r during the first Monte Carlo simulation and then calculate its mean. Then, we adjust the calculated value and we do as we have already shown.

Parallel Computations for European Options

The parallelization of European Options pricing computations is rather straightforward. If we would like more than one processes to compute values, all we have to do is to partition the problem in the respect of the loops we have decided that our simulation should do. When each and every process has finished computing the option value, we combine them by taking their sample mean.

For example, suppose we had to price a european option with Monte Carlo simulation and we had decided that we should loop for $1,000,000$ times. Then, suppose that we would like the computations to be done by four processes rather than one. In this point, we are facing a dilemma: it seems more right that each process should as the starting asset value S_0 for its computations from the final value of its previous process, but is this possible in a real parallel world? The answer is no. There is dependence between the computed asset values, as shown in (2.14). So, if we would to stick with this idea, then we would not be able to parallelize the problem, since its process should wait for its previous to end; actually this multi-processed solution would be less effective, since we would have the message passing overhead while we could do the same work with only one process.

The conclusion that we have reached from the previous idea is that each process should compute the option price from the beginning to the end. But, if we assigned each process $250,000$ loop times, then each process would have calculated the exact same value and, of course, it would be less accurate, since the option value would have been computed in $250,000$ loop times rather than $1,000,000$ loop times. Still, we have not found any other way to parallelize the problem. In order to increase accuracy, we could assign a different ϵ to each process. This way, each process would compute a different value and we could take the sample mean of these four values to be the option value. In this case, even though we do not solve the initial problem (the loop still lasts $250,000$ times), we have a better

sample for ϵ since we take more than one values. So, putting it all together, we may lose in one field, but we win in another one.

What we suggest is to increase the loop times for each process, since we have a significant speedup by multi-processing the computations as shown in Chapter 5. This way, we improve accuracy and the comparison between the serial and the parallel method is posed on a different basis. The parallelization of the problem in this case can be taken as an effort not only for speeding the computations but also for improving accuracy.

Next, we give the algorithm that has been used in the context of this thesis for European option pricing using parallel Monte Carlo methods. Please note that we have used the same number of iterations as in the serial method, so that the comparison would be in the same terms.

Algorithm 2 Parallel Monte Carlo Method

1: **For each process:**
2: $price = 0.0$;
3: $discount = e^{-r*T}$;
4: $times = M/nofProcesses$;
5: **for** $i = 0$ to $times$ **do**
6: $\quad \epsilon = getStandardNormalRandomNumber()$;
7: $\quad \delta S = S * r * T + S * \sigma * \epsilon * \sqrt{T}$;
8: $\quad S_{new} = S + \delta S$;
9: $\quad price+ = payoff(S_{new})$;
10: **end for**
11: $avgPrice = price/times$;
12: $derivativePrice = discount * avgPrice$;
13: **For process 0 after all processes have finished:**
14: $$optionPrice = \frac{\sum_{i=1}^{nofProcesses} derivativePrice_i}{nofProcesses};$$

If we used this method in a serial execution, that is we had only one process, then the cost would be linear $O(M)$, where M stands for the total steps that the method has to do to complete the computations. If we had more processes, say P, then the cost would be $O(M/P + 2PK)$, where K is the cost for sending or receiving one message, since all processes send their computed value to the first one, which computes the final option value.

In this case, the problem parallelization seems to be working very nice, because the computations can get very low and the communications cost is almost

insignificant as the problem scales. In Chapter 5 we give some experimental results that verify these assumptions.

Let us denote that [10, 8, 6] have given some ideas about how we will develop the source code and the algorithms.

2.6.2 American Options

Although American options are slightly different than the European options, it is much more difficult to evaluate them using Monte Carlo methods. First of all, let us note that their difference is the exercise date: European options have a fixed exercise date, which is the expiration, but American options can be exercised at any time until their expiration. As we will see next, this fact can be a check in applying Monte Carlo in American options.

However, some progress has been made. In this Section we will present the most common schemes that have been proposed until now. Several solutions have been given to this problem, but we will deal with three of them. At this point, we need to say that according to [17] the estimators proposed by Tilley [20] and Barraquand and Martineau [9] are both biased. So is the estimator proposed by Broadie and Glasserman [12] but they take that in mind and they try to find a solution to this problem. We will examine all these schemes in following Sections.

Tilley Scheme

Tilley discretizes time in N epochs and simulates different paths for the asset price for these epochs. The intrinsic value, that is the profit if the option is exercised, of an American option on path k at epoch t is defined as:

$$I(k,t) = \begin{cases} max[0, S(k,t) - X(t)] & \text{for a call option} \\ max[0, X(t) - S(k,t)] & \text{for a put option} \end{cases} \tag{2.15}$$

where $X(t)$ is the exercise price at epoch t.

Let $z(k,t)$ be the "exercise or hold" indicator variable which takes the value 0 if the option is not exercised at epoch t on path k and which takes the value 1 if the option is exercised at epoch t on path k. According to Tilley, in order to estimate the price of the option, we need to estimate this exercise-or-hold estimator z, given a finite sample o fR paths drawn from an arbitrage-free distribution of paths. Note that the latest epoch that the option can be exercised is the expiration date, where $z(k,N) = 1$ if and only if $I(k,N) > 0$. Next, we give the backward induction algorithm proposed by Tilley.

Algorithm 3 Tilley Monte Carlo for American Options

1: Reorder the stock price paths by stock price, from the lowest price to highest price for a call option or from highest price to lowest price for a put option. Reindex the paths from 1 to R according to the reordering.

2: For each path k, compute the intrinsic value $I(k,t)$ of the option.

3: Partition the set of R ordered paths into Q distinct bundles of P paths each. Assign the first P paths to the first bundle, the second P paths to the second bundle, and so on, and finally the last P paths to the Qth bundle. It is assumed that P and Q are integer factors of R.

4: For each path k, the option's "holding" value $H(k,t)$ is computed as the following mathematical expression taken over all paths in the bundle containing the path k:

$$H(k,t) = d(k,t)P^{-1} \sum_{\text{all j in bundle containing k}} V(j,t+1)$$

The variable $V(k,t)$ is fully defined in step 8 below. At epoch N, $V(k,N) = I(k,N)$ for all k

5: For each path, compare the holding value $H(k,t)$ to the intrinsic value $I(k,t)$ and decide "tentatively" whether to exercise or hold. Define an indicator variable $x(k,t)$ as follows:

$$x(k,t) = \begin{cases} 1 & \text{if } I(k,t) > H(k,t) \quad \text{Exrcise} \\ 0 & \text{if } H(k,t) \geq I(k,t) \quad \text{Hold} \end{cases}$$

6: Examine the sequence of 0's and 1's $x(k,t); k = 1,2,\ldots,R$. Determine a "sharp" boundary between the hold decision and the exercise decision as the start of the first string of 1's the length of which exceeds the length of every subsequent string of 0's. Let $k_*(t)$ denote the path index (in the sample as ordered in substep 1 above) of the leading 1 in such a string. The "transition zone" between hold and exercise is defined as the sequence of 0's and 1's that begins with the first 1 and ends with the last 0.

7: Define a new exercise or hold indicator variable $y(k,t)$ that incorporates the sharp boundary as follows:

$$y(k,t) = \begin{cases} 1 & \text{for } k \geq k_*(t) \\ 0 & \text{for } k < k_*(t) \end{cases}$$

8: For each path k, define the current value $V(k,t)$ of the option as follows:

$$V(k,t) = \begin{cases} I(k,t) & \text{if } y(k,t) = 1 \\ H(k,t) & \text{if } y(k,t) = 0 \end{cases}$$

After the algorithm has been processed backward from epoch N to epoch 1, the indicator variable $z(k,t)$ for $t < N$ is estimated as follows:

$$z(k,t) = \begin{cases} 1 & \text{if } y(k,t) = 1 \text{ and } y(k,s) = 0 \text{ for all } s < t \\ 0 & \text{otherwise} \end{cases}$$

For more details on the scheme suggested by Tilley, please refer to [20].

Barraquand - Martineau Scheme

Barraquand and Martineau [9] propose a scheme, where, contrarily to Tilley who partitions the state space, they partition the payoff space. In general, they first generate some sample paths, then they find some conditional probabilities and conditions and, finally, they use a backward integration algorithm. Next, we give more details about these steps.

We generate a given number M of sample paths for the underlying assets price process $X(t)$. In general, this can be done through direct numerical integration of the Ito equation:

$$\forall i \in [1,n], \quad \frac{dx_i}{x_i} = (r - d_x i)dt + \sum_{j=1}^{n} v_{ij} dw_j \tag{2.16}$$

where v is the volatility and w is a Wiener random variable. A simple explicit Euler scheme is given by:

$$x_i(t + \Delta t) = x_i(t) e^{(r - d_x i - \frac{1}{2}k_{ii})(X(t),t)\Delta t + \sum_{j=1}^{n} v_{ij}(X(t),t)\sqrt{\Delta t} z_j^t}$$

where z_j^t follow independent standard normal distributions for all j and t. For $d = T/\Delta t$ being the number of time steps in $[0,T]$, we must draw a total of $Mxdxn$ standard normal variates in order to generate M n-dimensional sample paths $X^1(t), \ldots, X^M(t)$ for all $t > 0$.

Once the M sample paths $X^1(t), \ldots, X^M(t)$ are computed, the number $\alpha_i(t)$ of samples crossing $P_i(t)$ and the number $bij(t)$ of samples moving from $P_i(t)$ to $P_j(t + \Delta t)$ are easily computed:

$$\begin{aligned} a_i(t) &= Card\{k \in [1,M], X^k(t) \in P_i(t)\} \\ b_{ij}(t) &= Card\{k \in [1,M], X^k(t) \in P_i(t) \text{and} X^k(t + \Delta t) \in P_j(t + \Delta t)\} \end{aligned}$$

Similarly, the sum $c_i(t)$ over of samples X^k of payoff values $f(X^k(t))$ is computed from:

$$c_i(t) = \sum_{\{k \in [1,M], X^k(t) \in P_i(t)\}} f(X^k(t))$$

By the law of large numbers, we have the following identities:

$$p_{ij}(t) = \lim_{M\to\infty} \frac{b_{ij}(t)}{a_i(t)} \quad f_i(t) = \lim_{M\to\infty} \frac{c_i(t)}{\alpha_i(t)}$$

Using the Monte Carlo estimates of the conditional probabilities and payoff expectations, an approximation of the American price can be then computed backwards in time using the following algorithm:

- At time T, the approximate SSAP price is initialised at:

$$C(i,T) = \frac{c_i(T)}{\alpha_i(T)}$$

- At time $T - \Delta t$, we can compute for all $i \in [1,k]$:

$$C(i,T-\Delta t) = e^{-r\Delta t} \max\left(\frac{c_i(T-\Delta t)}{\alpha_i(T-\Delta t)}, \sum_{j=1}^{k} C(j,T)\frac{b_{ij}(T-\Delta t)}{\alpha_i(T-\Delta t)}\right)$$

- The above procedure is then applied recursively, backwards in time, to compute all the prices $C(i,T-2\Delta t), C(i,T-3\Delta t), \ldots, C(1,0) = C_{SSAP}$

For more details in this scheme, please refer to [9].

Broadie and Glasserman Scheme

Broadie and Glasserman [12] propose a scheme based on simulated trees. They identify the bias problem as the main issue in Monte Carlo simulation for American options and they suggest a solution to this problem. They develop two estimators, one biased high and one biased low, but both convergent and asymptotically unbiased as the computational effort increases. Then, they obtain a valid confidence interval for the true value P by taking the upper confidence limit from the 'high' estimator and the lower confidence from the 'low' estimator. Next, we give the algorithm Broadie and Glasserman suggest in [12]:

Algorithm 4 Broadie and Glasserman Monte Carlo for American Options

▷ allocate storage

integer vector $w(j)$, **for** $j = 1$ to d by 1;
real matrix $v(i,j)$, **for** $i = 1$ to b by 1, $j = 1$ to d by 1;

▷ initialise parameters

$v(1,1) = S$; $w(1) = 1$;
for $j = 2$ to d by 1 **do**
    $v(1,j)$ = 'state variable';
    $w(j) = 1$;
end for

▷ process tree

$j = d$;
while $j > 0$ **do**
    case 1: ($j = d$ and $w(j) < b$)
    $v(w(j),j)$ = 'node value';
    $v(w(j)+1,j)$ = 'state variable';
    $w(j) = w(j) + 1$;
    end case 1
    case 2: ($j = d$ and $w(j) = b$)
    $v(w(j),j)$ = 'node value';
    $w(j) = 0$;
    $j = j - 1$;
    end case 2
    case 3: ($j < d$ and $w(j) < b$)
    $v(w(j),j)$ = 'node value';

    if $j > 1$ **then**
        $v(w(j)+1,j)$ = 'state variable';
        $w(j) = w(j) + 1$;
        for $i = j + 1$ to d by 1 **do**
            $v(1,i)$ = 'state variable';
            $w(i) = 1$;
        end for
        $j = d$;
    else
        $j = 0$;
    end if
    end case 3
    case 4: ($j < d$ and $w(j) = b$)
    $v(w(j),j)$ = 'node value';
    $w(j) = 0$;
    $j = j - 1$;
    end case 4
end while

▷ return tree estimate

'tree estimate' = $v(1,1)$;

Chapter 3

Partial Differential Equations

3.1 Introduction

The numerical methods for solving partial differential equations in finance are not widely used. The reason is the existence of several probabilistic values, for which Monte Carlo methods are usually preferred. However, if we could discretize the problem then we could use some algorithms to solve them which could be proved to be much more efficient. Furthermore, this technique gives us useful information during the solving process, such as the option price for all values of the maturity and for all spot prices. Finally, it is useful for computing the so-called "Greeks".

The partial differential equations in financial problems are sometimes posed in a bounded domain but most of the times in an unbounded domain; in the last case we must find the suitable boundary conditions and use appropriate numerical approximations. These partial differential equations are usually of parabolic type and the numerical methods used to solve them should be sufficiently fast and accurate.

Here, we are going to present the finite difference and the finite element method for solving partial differential equations applied in option pricing.

3.2 Methods for Solving Partial Differential Equations

3.2.1 Finite Difference Method

The basic idea here is to replace the partial derivatives by approximations that arise from Taylor series expansions of functions near the points of interest. So, if

we had to use finite differences for the partial derivative

$$\frac{\partial u}{\partial t}(x,t) = \lim_{\delta t \to 0} \frac{u(x,t+\delta t) - u(x,t)}{\delta t}$$

we would use the Taylor's theorem

$$u(x,t_0+\delta t) = u(x,t_0) + \frac{\partial u}{\partial t}(x,t_0)\delta t + O(\delta t^2)$$

and then we would have

$$\frac{\partial u}{\partial t}(x,t) = \frac{u(x,t+\delta t) - u(x,t)}{\delta t} + O(\delta t)$$

This particular finite difference approximation is called forward difference, since the differencing is in the forward t direction. There is also the backward difference approximation, which is defined as follows

$$\frac{\partial u}{\partial t}(x,t) = \frac{u(x,t) - u(x,t-\delta t)}{\delta t} + O(\delta t)$$

and the central difference approximation which is defined as follows

$$\frac{\partial u}{\partial t}(x,t) = \frac{u(x,t+\delta t) - u(x,t-\delta t)}{2\delta t} + O((\delta t)^2)$$

The central differences are more accurate for small δt than the two other schemes.

Next, we show the three most common methods in solving partial differential equations using the finite-differences scheme. We are going to use the diffusion equation in order to illustrate these methods, as the problems that arise in derivative pricing can be solved either using the diffusion equation, or solving directly the Black-Scholes[2] equation; the last though is similar to the first. The diffusion equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad x \in (x_{low}, x_{high}), \quad t > 0 \tag{3.1}$$

with homogeneous Dirichlet boundary conditions

$$u(0,t) = u(1,t) = 0 \quad and \quad u(x,0) = u_0(x)$$

We are going to partition the x dimension in $x_0, x_1, \ldots, x_N$ and the t dimension in $t_0, t_1, \ldots, t_M$. So, each point will be represented by

$$u(x_n, t_m) = u_n^m$$

3.2.1.1 Explicit Method

By using a forward difference for $\partial u/\partial t$ and a symmetric central difference for $\partial^2 u/\partial t^2$, we find that the diffusion equation becomes

$$\frac{u_n^{m+1} - u_n^m}{\delta t} + O(\delta t) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2) \tag{3.2}$$

Ignoring the terms $O(\delta t)$ and $O((\delta x)^2)$, we can rearrange the above equation to give

$$u_n^{m+1} = \alpha u_{n+1}^m + (1 - 2\alpha)u_n^m + \alpha u_{n-1}^m \tag{3.3}$$

where

$$\alpha = \frac{\delta t}{(\delta x)^2}$$

There is a stability problem when we are using explicit finite differences to solve partial differential equations. This problem arises because we are using finite precision computer arithmetic to solve them, which introduces rounding errors into their numerical solutions. In our example, the system (3.3) is said to be stable if these rounding errors are not magnified at each iteration; in any other case it is said to be unstable. More precisely, the system (3.3) is said to be:

- stable, if $0 < \alpha \leq \frac{1}{2}$
- unstable, if $\alpha > \frac{1}{2}$

It can be shown that the numerical solution of the finite difference equations converges to the exact solution of the diffusion equation as $\delta x \to 0$ and $\delta t \to 0$, in the sense that

$$u_n^m \to u(n\delta x, m\delta t)$$

if and only if the explicit finite difference method is stable.

Next, we give an example of an algorithm which gives the solution for the diffusion equation using the explicit finite-difference scheme:

Let us denote that N^- and N^+ are the lower and the higher bound respectively for x and we have that:

$$N^-\delta x \leq x \leq N^+\delta x$$

So, the boundary conditions are given by

$$u_{-\infty}(N^-\delta x, m\delta t)$$

and

$$u_{\infty}(N^+\delta x, m\delta t)$$

Note that we use $u_{-\infty}$ and u_{∞} to indicate that we take these values to be the lower and the upper bound.

Algorithm 5 Explicit Finite-Differences

```
1: a = δt/δx^2;
2: for n = N^- to N^+ do
3:     oldu[n] = payoff(n * δx);
4: end for
5: for m = 1 to M do
6:     τ = m * δt;
7:     newu[N^-] = u_{-∞}(N^- * δx, τ);
8:     newu[N^+] = u_{+∞}(N^+ * δx, τ);
9:     for n = N^- + 1 to N^+ do
10:        newu[n] = oldu[n] + α * (oldu[n - 1] - 2 * oldu[n] + oldu[n + 1]);
11:    end for
12:    for n = N^- to N^+ do
13:        oldu[n] = newu[n];
14:    end for
15: end for
16: for n = N^- to N^+ do
17:    values[n] = oldu[n];
18: end for
```

3.2.1.2 Implicit Method

Implicit finite differences do not face the stability problems that arise in explicit finite differences. So, we can use a large number of x points without having to take very small time-steps.

For the diffusion equation (3.1), the implicit finite differences scheme uses the backward difference approximation for $\partial u/\partial t$ term and the symmetric central difference approximation for $\partial^2 u/\partial x^2$ term. This leads to the equation

$$\frac{u_n^m - u_n^{m-1}}{\delta t} + O(\delta t) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2) \tag{3.4}$$

where $n = N^-, \ldots, N^+$ and $m = 0, \ldots, M$. Following the same procedure as with the explicit scheme, we rearrange the above equation to

$$-\alpha u_{n-1}^m + (1 + 2\alpha)u_n^m - \alpha u_{n+1}^m = u_n^{m-1} \tag{3.5}$$

The implicit finite difference method leads us to a system which we have to solve. For this particular problem, the linear system is of the form $Ax = b$, where A is a $(N^- + N^+ - 1)x(N^- + N^+ - 1)$ matrix and both x and b are $(N^- + N^+ - 1)x1$

vectors:

$$\begin{pmatrix} 1+2\alpha & -\alpha & 0 & \dots & 0 \\ -\alpha & 1+2\alpha & -\alpha & \dots & 0 \\ 0 & -\alpha & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & -\alpha \\ 0 & 0 & \dots & -\alpha & 1+2\alpha \end{pmatrix} \begin{pmatrix} u^m_{N^-+1} \\ \vdots \\ u^m_0 \\ \vdots \\ u^m_{N^+-1} \end{pmatrix}$$
$$= \begin{pmatrix} u^{m-1}_{N^-+1} \\ \vdots \\ u^{m-1}_0 \\ \vdots \\ u^{m-1}_{N^+-1} \end{pmatrix} + \alpha \begin{pmatrix} u^m_{N^-} \\ 0 \\ \vdots \\ 0 \\ u^m_{N^+} \end{pmatrix} = \begin{pmatrix} b^m_{N^-+1} \\ \vdots \\ b^m_0 \\ \vdots \\ b^m_{N^+-1} \end{pmatrix} \tag{3.6}$$

where

$$u^m_{N^-} = u_{-\infty}(N^-\delta x, m\delta t), \quad 0 < m \leq M$$
$$u^m_{N^+} = u_{\infty}(N^+\delta x, m\delta t), \quad 0 < m \leq M$$

and

$$N^-\delta x \leq x \leq N^+\delta x$$

In order to solve the system (3.6) we choose a direct or an iterative method of our choice.

Next, we give an example of an algorithm which gives the solution for the diffusion equation using the implicit finite-difference scheme and the SOR method.

Algorithm 6 *SORSolver*$(u, b, N^-, N^+, \alpha, \omega, eps, loops)$

```
1:  loops = 0;
2:  repeat
3:      error = 0.0;
4:      for n = N^- + 1 to N^+ do
5:          y = (b[n] + α * (u[n − 1] + u[n + 1]))/(1 + 2 * α);
6:          y = u[n] + ω * (y − u[n]);
7:          error+ = (u[n] − y)^2;
8:          u[n] = y;
9:      end for
10:     + + loops;
11: until error > eps;
        return loops;
```

Algorithm 7 Implicit Finite-Differences

$a = \delta t/\delta x^2$;
$eps = 1.0e-8$;
$\omega = 1.0$;
$d\omega = 0.05$;
$oldloops = 10000$;
for $n = N^-$ to N^+ **do**
    $values[n] = payoff(n * \delta x)$;
end for
for $m = 1$ to M **do**
    $\tau = m * \delta t$;
    for $n = N^- + 1$ to N^+ **do**
        $b[n] = values[n]$;
    end for
    $values[N^-] = u_{-\infty}(N^- * \delta x, \tau)$;
    $values[N^+] = u_{+\infty}(N^+ * \delta x, \tau)$;
    $SORSolver(values, b, N^-, N^+, \alpha, \omega, eps, loops)$;
    if $loops > oldloops$ **then**
        $d\omega* = -1.0$;
    end if
    $\omega+ = d\omega$;
    $oldloops = loops$;
end for

3.2.1.3 Crank-Nicolson Method

The Crank-Nicolson scheme is used to overcome the stability limitations of the explicit scheme and to have $O((\delta t)^2)$ rate of convergence to the solution of the partial differential equation -the rate of convergence for the implicit scheme is $O(\delta t)$.

This method is based on both explicit and implicit finite differences methods, by taking their average value. So, for the diffusion equation (3.1), we take the forward difference approximation for the explicit scheme

$$\frac{u_n^{m+1} - u_n^m}{\delta t} + O(\delta t) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2)$$

and the backward difference approximation for the implicit scheme

$$\frac{u_n^m - u_n^{m-1}}{\delta t} + O(\delta t) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2).$$

Finally, we take the average of the two above equations and we have

$$\frac{u_n^{m+1} - u_n^m}{\delta t} + O(\delta t) = \frac{1}{2}\left(\frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{(\delta x)^2}\right) + O((\delta x)^2) \quad (3.7)$$

Following the same procedure as with the two previous methods, we are lead to the Crank-Nicolson scheme:

$$u_n^{m+1} - \frac{1}{2}\alpha(u_{n-1}^{m+1} - 2u_n^{m+1} + u_{n+1}^{m+1}) = u_n^m + \frac{1}{2}\alpha(u_{n-1}^m - 2u_n^m + u_{n+1}^m) \quad (3.8)$$

where

$$\alpha = \frac{\delta t}{(\delta x)^2}$$

Finally, the problem reduces to calculating

$$Z_n^m = (1-\alpha)u_n^m + \frac{1}{2}\alpha(u_{n-1}^m + u_{n+1}^m) \quad (3.9)$$

and then solving

$$(1+\alpha)u_n^{m+1} - \frac{1}{2}\alpha(u_{n-1}^{m+1} + u_{n+1}^{m+1}) = Z_n^m \quad (3.10)$$

where $n = N^-, \ldots, N^+$ and $m = 0, \ldots, M$.

Working as above, we reduce our problem to solving the linear system

$$Cu^{m+1} = b^m \quad (3.11)$$

where the $(N^- + N^+ - 1)x(N^- + N^+ - 1)$ matrix C is given by

$$C = \begin{pmatrix} 1+\alpha & -\frac{1}{2}\alpha & 0 & \ldots & 0 \\ -\frac{1}{2}\alpha & 1+\alpha & -\frac{1}{2}\alpha & & \vdots \\ 0 & -\frac{1}{2}\alpha & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\frac{1}{2}\alpha \\ 0 & 0 & & -\frac{1}{2}\alpha & 1+\alpha \end{pmatrix} \quad (3.12)$$

and the $(N^- + N^+ - 1)x1$ vectors u^{m+1} and b^m are given by

$$u^{m+1} = \begin{pmatrix} u_{N^-+1}^{m+1} \\ \vdots \\ u_0^{m+1} \\ \vdots \\ u_{N^+-1}^{m+1} \end{pmatrix}, b^m = \begin{pmatrix} Z_{N^-+1}^m \\ \vdots \\ Z_0^m \\ \vdots \\ Z_{N^+-1}^m \end{pmatrix} + \frac{1}{2}\alpha \begin{pmatrix} u_{N^-}^{m+1} \\ 0 \\ \vdots \\ 0 \\ u_{N^+}^{m+1} \end{pmatrix} \quad (3.13)$$

From this point on, we can choose a direct or an iterative method to solve the linear system. In our case, we will choose an iterative method, which we will parallelize.

The Crank-Nicolson is both stable and convergent for all values of $\alpha > 0$.

Algorithm 8 Crank-Nicolson

$\alpha = \delta t/\delta x^2$;
$\alpha_2 = \alpha/2$;
$\omega = 1.0$;
$d\omega = 0.05$;
$oldloops = 10000$;
for $n = N^-$ to N^+ **do**
    $val[n] = payoff(n * \delta x)$;
end for
for $m = 1$ to M **do**
    $\tau = m * \delta t$;
    for $n = N^- + 1$ to N^+ **do**
        $b[n] = (1 - \alpha) * val[n] + \alpha_2 * (val[n + 1] + val[n - 1])$;
    end for
    $val[N^-] = u_{-\infty}(N^- * \delta x, \tau)$;
    $val[N^+] = u_{+\infty}(N^+ * \delta x, \tau)$;
    $SORSolver(val, b, N^-, N^+, \alpha_2, \omega, eps, loops)$;
    if $loops > oldloops$ **then**
        $d\omega* = -1.0$;
    end if
    $\omega+ = d\omega$;
    $oldloops = loops$;
end for

Next, we give an example of an algorithm which gives the solution for the diffusion equation using the Crank-Nicolson scheme and the SOR method.

3.3 Option Pricing

After analysing the methods above for solving partial differential equations, we will show how they can be applied in option pricing.

3.3.1 European Options

We will use the Black-Scholes model for european options:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{3.14}$$

where V is the price of the option, $t > 0$ is the current time, σ is the asset price volatility, $S > 0$ is the asset price and r the constant interest rate.

The boundary conditions for the equation (3.14) are:

- At maturity, $t = T$, the final condition comes from the no arbitrage principle. Thus, the value of a call option will be the payoff:

$$C(S,T) = \max(S - E, 0) \tag{3.15}$$

and the value of a put option will be the payoff:

$$P(S,T) = \max(E - S, 0) \tag{3.16}$$

where E is the exercise price of the option.

- At zero asset price, $S = 0$, we can see that, for a call option, the asset price can never change, as $dS = 0$. Thus, the payoff is also zero and the call option is worthless:

$$C(0,t) = 0 \tag{3.17}$$

In case we have a put option, since the asset price cannot change, the price of the option must equal the exercise price at expiry. So, we take the discounted value of E with constant interest rate and we have:

$$P(0,t) = Ee^{-r(T-t)} \tag{3.18}$$

For a time-dependent interest rate, we have:

$$P(0,t) = Ee^{-\int_t^T r(\tau)d\tau} \tag{3.19}$$

- As the asset price increases without bound, $S \to \infty$, it becomes more likely that the call option will be exercised and the exercise price becomes less important. Thus, as $S \to \infty$, the value of the option becomes that of the asset and we write:

$$C(S,t) \sim S \quad as \quad S \to \infty \tag{3.20}$$

On the contrary, if $S \to \infty$, a put option is unlikely to be exercised and we write:

$$P(S,t) \to 0 \quad as \quad S \to \infty \tag{3.21}$$

3.3.1.1 Finite Difference Method

Here, we will introduce the finite difference method in european options pricing as illustrated in [22]. Our goal is to use the three methods analysed above, to solve the Black-Scholes partial differential equation and evaluate the options. First, we present the discretization of (3.14) with respect to the variable S, i.e. the asset price. We divide the interval $[0, S_{\max}]$ into N intervals of length $\delta S = S_{\max}/N$ and we approximate the derivatives with finite differences. So, a possible discretization scheme for (3.14) is:

$$\frac{\partial V_n}{\partial t} + rS_n\frac{V_{n+1} - V_{n-1}}{2\delta S} + \frac{1}{2}\sigma^2 S_n^2\frac{V_{n+1} - 2V_n + V_{n-1}}{\delta S^2} - rV_n = 0 \tag{3.22}$$

where $S_n = n\delta S$ denotes the n-th discretization point and $V_n(t)$ is intended to be an approximation of $V(t, S_n)$.

Then, we divide the time interval $[0, T]$ into M intervals of length $\delta t = T/M$ and we replace the time derivative by a finite difference. For each numerical method illustrated above, we have:

3.3.1.1.1 Explicit Finite Difference Method

Using the explicit finite difference scheme and, since we have end and not initial condition values, equation (3.22) becomes:

$$\frac{V_n^{m+1} - V_n^m}{\delta t} + rS_n \frac{V_{n+1}^{m+1} - V_{n-1}^{m+1}}{2\delta S} + \frac{1}{2}\sigma^2 S_n^2 \frac{V_{n+1}^{m+1} - 2V_n^{m+1} + V_{n-1}^{m+1}}{\delta S^2} - rV_n^{m+1} = 0. \tag{3.23}$$

where $n = 0, \ldots, N$ and $m = 0, \ldots, M$.

After working on the above equation, we end up to the equation:

$$V_n^m = A_n V_{n-1}^{m+1} + B_n V_n^{m+1} + C_n V_{n+1}^{m+1} \tag{3.24}$$

where

$$A_n = \frac{1}{2}\left(\sigma^2 n^2 - nr\right)\delta t, \tag{3.25}$$

$$B_n = 1 - \left(\sigma^2 n^2 + r\right)\delta t, \tag{3.26}$$

and

$$C_n = \frac{1}{2}\left(\sigma^2 n^2 + nr\right)\delta t \tag{3.27}$$

Now, we have to adjust the boundary conditions, in order to solve the problem numerically. Based on the beginning of this Section, we have the following boundary conditions for a european call option:

- At expiry, where $T = M\delta t$, we know that

$$V_n^M = \max(n\delta S - E, 0). \tag{3.28}$$

- At $S = 0$, we have that $n = 0$ and, so

$$A_0 = 0 \quad B_0 = 1 - r\delta t \quad C_0 = 0 \tag{3.29}$$

  and the scheme reduces to

$$(1 - r\delta t)V_0^{m+1} = V_0^m. \tag{3.30}$$

- When $S \to \infty$, say $S = S^*$, we have than $n = N$ and the equation becomes:

$$V_N^m = A_N V_{N-1}^{m+1} + B_N V_N^{m+1} + C_N V_{N+1}^{m+1} \tag{3.31}$$

There is a problem with equation (3.31): The value of V_{N+1}^{m+1} cannot be defined. In order to calculate its value, we consider the following: For most realistic options, we have that:

$$\lim_{S \to \infty} \frac{\partial^2 V}{\partial S^2} \to 0 \tag{3.32}$$

We approximate:

$$\frac{\partial^2 V}{\partial S^2}(S^*, t) \sim 0 \tag{3.33}$$

which becomes (using central differences):

$$V_{N+1}^{m+1} = 2V_N^{m+1} - V_{N-1}^{m+1}. \tag{3.34}$$

After substituting in equation (3.31) we have that:

$$V_N^m = \hat{A}_N V_{N-1}^{m+1} + \hat{B}_N V_N^{m+1} \tag{3.35}$$

where

$$\hat{A}_N = -Nr\delta t \tag{3.36}$$

and

$$\hat{B}_N = 1 + (N-1)r\delta t \tag{3.37}$$

Now that we have set up both the equation and the boundary conditions, we need to solve the system of equations that is being formed. This system consists of N linear equations. Since we wish to parallelize the problem, it would better be $N = 2^x$, as we usually use a number of processes which is a power of 2. Next we will give an example illustrating how we solve this system.

In this example, we consider four processes and eight discretization points ($N = 8$). So, we have the following linear system:

$$\begin{cases} V_0^m = B_0 V_0^{m+1} \\ V_1^m = A_1 V_0^{m+1} + B_1 V_1^{m+1} + C_1 V_2^{m+1} \\ V_2^m = A_2 V_1^{m+1} + B_2 V_2^{m+1} + C_2 V_3^{m+1} \\ V_3^m = A_3 V_2^{m+1} + B_3 V_3^{m+1} + C_3 V_4^{m+1} \\ V_4^m = A_4 V_3^{m+1} + B_4 V_4^{m+1} + C_4 V_5^{m+1} \\ V_5^m = A_5 V_4^{m+1} + B_5 V_5^{m+1} + C_5 V_6^{m+1} \\ V_6^m = A_6 V_5^{m+1} + B_6 V_6^{m+1} + C_6 V_7^{m+1} \\ V_7^m = \hat{A}_7 V_6^{m+1} + \hat{B}_7 V_7^{m+1} \end{cases} \tag{3.38}$$

Let us denote that the first and the last equations have been imposed by the corresponding boundary conditions.

Let M be the coefficients matrix, defined as follows:

$$M = \begin{bmatrix} B_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_2 & C_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_3 & B_3 & C_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_4 & B_4 & C_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & A_5 & B_5 & C_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_6 & B_6 & C_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & \hat{A}_7 & \hat{B}_7 \end{bmatrix} \tag{3.39}$$

and let

$$V^{m+1} = \begin{bmatrix} V_0^{m+1} \\ V_1^{m+1} \\ V_2^{m+1} \\ V_3^{m+1} \\ V_4^{m+1} \\ V_5^{m+1} \\ V_6^{m+1} \\ V_7^{m+1} \end{bmatrix} \quad and \quad V^m = \begin{bmatrix} V_0^m \\ V_1^m \\ V_2^m \\ V_3^m \\ V_4^m \\ V_5^m \\ V_6^m \\ V_7^m \end{bmatrix} \tag{3.40}$$

So, now the system has the form:

$$V^m = MV^{m+1} \tag{3.41}$$

From equation (3.28) we know the value of V^{m+1} at expiry, so we begin to solve the system for $m = M, M-1, M-2, \ldots, 1$.

Since we have four processes, we will assign two lines of each matrix or vector to each process. So, for example, the second process will hold the following values:

$$M = \begin{bmatrix} 0 & A_2 & B_2 & C_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_3 & B_3 & C_3 & 0 & 0 & 0 \end{bmatrix},$$
$$V^m = \begin{bmatrix} V_2^m \\ V_3^m \end{bmatrix} \quad and \quad V^{m+1} = \begin{bmatrix} V_2^{m+1} \\ V_3^{m+1} \end{bmatrix} \tag{3.42}$$

So, every process, just has to calculate the right part of (3.41). The only problem in doing so is that it is possible for a process not to have all necessary values to do these calculations, as in our example. Here, process 1 needs V_1^{m+1} from process 0 to calculate V_2^m and V_4^{m+1} from process 2 to calculate the value of V_3^m. The MPI framework gives us the answer to this problem.

At this point, it worth mentioning that the communication cost is rather significant. Each process needs to send and receive two messages, except the first

and the last process that need to send and receive one message. But, although the communication is often, it is small and fixed (each process exchanges one message with its neighbours) no matter how many values need to be computed or how many processes are up, solving the problem.

Each process calculates the vector V_n^m and sends the values to the previous or the next processes, if needed. When a process wants to calculate the new values for vector V_n^m, it first receives any values sent by other processes, if needed. Then, it has all data needed to proceed with the calculations and the solution of the system. For more details, see the source code for european option pricing using the explicit finite difference scheme.

Next we give the algorithm on which we have been based to price European options using Explicit Finite-Differences in the context of this thesis.

The cost for this method is $O(M * N)$ for one process and $O(M * N/P)$ for P processes considering steady cost $O(1)$ for the message passing, that is we have a very good hardware support. If not, then, depending on the system's specifications, a respective logarithmic factor will be added in the cost. Let us denote that the processes exchange four messages, except the first and the last ones that exchange only two messages. Based on these observations, one would expect the parallel execution to be faster than the serial one, only if the communication cost is so small, that the total cost does not become grater than $O(M * N)$. Unfortunately, this is hard to achieve without the best hardware support for communications, as shown in Chapter 5.

3.3.1.1.2 Implicit Finite Difference Method

Using the implicit finite difference scheme and, since we have end and not initial condition values, equation (3.22) becomes:

$$\frac{V_n^{m+1} - V_n^m}{\delta t} + rS_n \frac{V_{n+1}^m - V_{n-1}^m}{2\delta S} + \frac{1}{2} \frac{V_{n+1}^m - 2V_n^m + V_{n-1}^m}{\delta S^2} - rV_n^m = 0 \quad (3.43)$$

After working on the equation above, we end up to the equation:

$$V_n^{m+1} = A_n V_{n-1}^m + B_n V_n^m + C_n V_{n+1}^m \quad (3.44)$$

where

$$A_n = \frac{1}{2}(nr - \sigma^2 n^2)\delta t, \quad (3.45)$$

$$B_n = 1 + (\sigma^2 n^2 + r)\delta t, \quad (3.46)$$

Algorithm 9 Parallel Explicit Finite-Differences

```
1: For each process:
2: for n = 0 to processChunk do
3:     currentPos = processID * processChunk + n;
4:     oldV[n] = payoff(currentPos * δx);
5: end for
6: if processID == 0 then
7:     Send(oldV[processChunk − 1], nextProcess);
8: end if
9: if (processID > 0)  AND  (processID < nofProcesses − 1) then
10:    Send(oldV[0], previousProcess);
11:    Send(oldV[processChunk − 1], nextProcess);
12: end if
13: if processID == nofProcesses − 1 then
14:    Send(oldV[0], previousProcess);
15: end if
16: for m = M − 1 to 0 do
17:    for n = 0 to processChunk do
18:        if processID == 0 then
19:            if n == 0 then
20:                prevValue = 0.0;
21:            else
22:                prevValue = oldV[n − 1];
23:            end if
24:            if n == processChunk − 1 then
25:                Recv(nxtValue, nextProcess);
26:            else
27:                nxtValue = oldV[n + 1];
28:            end if
29:            V[n] = A_n * prevValue + B_n * oldV[n] + C_n * nxtValue;
30:            if n == processChunk − 1 then
31:                Send(V[n], nextProcess);
32:            end if
33:        else if (processID > 0)  AND  (processID < nofProcesses − 1)
   then
34:            if n = 0 then
35:                Recv(prevValue, previousProcesses);
36:            else
37:                prevValue = oldV[n − 1];
38:            end if
39:            if n == processChunk − 1 then
40:                Recv(nxtValue, nextProcess);
41:            else
42:                nxtValue = oldV[n + 1];
43:            end if
44:            V[n] = A_n * prevValue + B_n * oldV[n] + C_n * nxtValue;
45:            if n == 0 then
46:                Send(V[n], previousProcess);
47:            end if
48:            if n == processChunk − 1 then
49:                Send(V[n], nextProcess);
50:            end if
```

Algorithm 10 Parallel Explicit Finite-Differences Continued

```
51:         else if processID == nofProcesses − 1 then
52:             if n == 0 then
53:                 Recv(prevValue, previousProcess);
54:             else
55:                 prevValue = oldV[n − 1];
56:             end if
57:             if n == processChunk − 1 then
58:                 nxtValue = 0.0;
59:             else
60:                 nxtValue = oldV[n + 1];
61:             end if
62:             V[n] = A_n * prevValue + B_n * oldV[n] + C_n * nxtValue;
63:             if n == 0 then
64:                 Send(V[n], previousProcess);
65:             end if
66:         end if
67:     end for
68:     for n = 0 to processChunk do
69:         oldV[n] = V[n];
70:     end for
71: end for
72: for n = 0 to processChunk do
73:     derivativePrice[n] = oldV[n];
74: end for
```

and

$$C_n = -\frac{1}{2}(nr + \sigma^2 n^2)\delta t \tag{3.47}$$

Once again, based on the boundary conditions we gave at the beginning of this Section, we have the following conditions to solve the problem of pricing a european call option using the implicit finite difference scheme:

- At expiry, $T = M\delta t$, we know that

$$V_n^M = \max(n\delta S - E, 0). \tag{3.48}$$

- At $S = 0$, we have that $n = 0$ and, so

$$A_0 = 0 \quad B_0 = 1 + r\delta t \quad C_0 = 0 \tag{3.49}$$

  and the scheme reduces to

$$(1 + r\delta t)V_0^m = V_n^{m+1}. \tag{3.50}$$

- When $S \to \infty$, say $S = S^*$, we have that $n = N$ and the equation becomes:

$$V_N^{m+1} = A_N V_{N-1}^m + B_N V_N^m + C_N V_{N+1}^m \tag{3.51}$$

Like before, there is a problem with equation (3.51): We cannot calculate the value of V_{N+1}^m. So, we follow the same procedure, as above, and we have:

$$\lim_{S \to \infty} \frac{\partial^2 V}{\partial S^2} \to 0$$

By using the approximation

$$\frac{\partial^2 V}{\partial S^2}(S^*, t) \sim 0$$

we end up to the equation

$$V_{N+1}^m = 2V_N^m - V_{N-1}^m. \tag{3.52}$$

After substituting in equation (3.51), we have that

$$V_N^{m+1} = \hat{A}_N V_{N-1}^m + \hat{B}_N V_N^m \tag{3.53}$$

where

$$\hat{A}_N = nr\delta t \tag{3.54}$$

and

$$\hat{B}_N = 1 + r(1-n)\delta t \tag{3.55}$$

Now that everything is given, we can proceed to out example in order to demonstrate the parallel solution process. Once again, we consider four processes and eight discretization points ($N = 8$). So, we have to solve the following linear system:

$$\begin{cases} B_0 V_0^m = V_0^{m+1} \\ A_1 V_0^m + B_1 V_1^m + C_1 V_2^m = V_1^{m+1} \\ A_2 V_1^m + B_2 V_2^m + C_2 V_3^m = V_2^{m+1} \\ A_3 V_2^m + B_3 V_3^m + C_3 V_4^m = V_3^{m+1} \\ A_4 V_3^m + B_4 V_4^m + C_4 V_5^m = V_4^{m+1} \\ A_5 V_4^m + B_5 V_5^m + C_5 V_6^m = V_5^{m+1} \\ A_6 V_5^m + B_6 V_6^m + C_6 V_7^m = V_6^{m+1} \\ \hat{A}_7 V_6^m + \hat{B}_7 V_7^m = V_7^{m+1} \end{cases} \tag{3.56}$$

The first and the last equations have been imposed by the corresponding boundary conditions.

Let M be the coefficients matrix, defined as follows:

$$\begin{bmatrix} B_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_2 & C_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_3 & B_3 & C_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_4 & B_4 & C_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & A_5 & B_5 & C_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_6 & B_6 & C_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & \hat{A}_7 & \hat{B}_7 \end{bmatrix} \tag{3.57}$$

and let

$$V^{m+1} = \begin{bmatrix} V_0^{m+1} \\ V_1^{m+1} \\ V_2^{m+1} \\ V_3^{m+1} \\ V_4^{m+1} \\ V_5^{m+1} \\ V_6^{m+1} \\ V_7^{m+1} \end{bmatrix} \quad and \quad V^m = \begin{bmatrix} V_0^m \\ V_1^m \\ V_2^m \\ V_3^m \\ V_4^m \\ V_5^m \\ V_6^m \\ V_7^m \end{bmatrix} \tag{3.58}$$

Now we have to solve the system:

$$MV^m = V^{m+1} \tag{3.59}$$

From equation (3.48) we know the value of the option at expiry, i.e. V_n^{m+1}, so we solve the system for $m = M, M-1, \ldots, 1$.

Since we have four processes to solve this system, we assign $N/4 = 2$ lines of each matrix to each process. So, for example, the second process will hold the values:

$$M = \begin{bmatrix} 0 & A_2 & B_2 & C_2 & 0 & 0 & 0 \\ 0 & 0 & A_3 & B_3 & C_3 & 0 & 0 \end{bmatrix},$$
$$V^m = \begin{bmatrix} V_2^m \\ V_3^m \end{bmatrix} \quad and \quad V^{m+1} = \begin{bmatrix} V_2^{m+1} \\ V_3^{m+1} \end{bmatrix} \tag{3.60}$$

Each process has to solve a smaller system of equations. In order to do so, we can choose either direct or iterative methods. Due to the nature of the parallelization we have chosen, iterative methods are strongly recommended. In the context of this thesis, Jacobi method has been used. In more details, we have that:

$$V_2^{m,(k+1)} = \frac{V_2^{m+1,(k)} - A_2 V_1^{m,(k)} - C_2 V_3^{m,(k)}}{B_2} \tag{3.61}$$

and

$$V_3^{m,(k+1)} = \frac{V_3^{m+1,(k)} - A_3 V_2^{m,(k)} - C_3 V_4^{m,(k)}}{B_3} \tag{3.62}$$

where k is the Jacobi method step. For $k = 0$, we have that $V_n^{m,(k)}$ is equal to V_n that was evaluated in the previous step.

As we can easily see, for each value we want to calculate, we need both the previous and the next values. These values may not always belong to the same process. In our example, we need either a value from the previous process (for V_2^m) or a value from the next process (for V_3^m). Our processes exchange values using the MPI framework. So, each time a new value is being calculated, then it is sent to the process that will need it, or that is already waiting for it.

This way, we can parallelize our problem in a higher level and each process is dealing with a same problem as the original; only smaller.

Next, we give the algorithm for European option pricing using parallel implicit finite-differences scheme. We also give the algorithm for our parallel version of the Jacobi method, used by the implicit scheme. Note that for the Jacobi method, we are solving the system $Ax = b$ where the matrix A splits in $A = L + D + U$. Note that L is the lower triangle part of A, D is the diagonal part of A and U is the upper triangle part of A.

Algorithm 11 Parallel Jacobi Method

```
1:  For each process:
2:  sent = false;
3:  for k = 1 to iterations do
4:      for i = 0 to processChunk do
5:          realPos = processID * processChunk + i;
6:          if processID == 0 then
7:              if i == 0 then
8:                  prevValue = 0.0;
9:                  if processChunk > 1 then
10:                     nxtValue = x[i + 1];
11:                     x_new[i] = (b[i] − L[i] * prevValue − U[i] * nxtValue)/D[i];
12:                 else
13:                     if (k == 1)  AND  (!sent) then
14:                         sent = true;
15:                         Send(b[i], nextProcess);
16:                     end if
17:                     Recv(nxtValue, nextProcess);
18:                     x_new[i] = (b[i] − L[i] * prevValue − U[i] * nxtValue)/D[i];
19:                     if k! = iterations then
20:                         Send(x_new[i], nextProcess);
21:                     end if
22:                 end if
23:             else if i == processChunk − 1 then
24:                 if (k == 1)  AND  (!sent) then
25:                     sent = true;
26:                     Send(b[i], nextProcess;
27:                 end if
28:                 prevValue = x[i − 1];
29:                 Recv(nxtValue, nextProcess);
30:                 x_new[i] = (b[i] − L[i] * prevValue − U[i] * nxtValue)/D[i];
31:                 if k! = iterations then
32:                     Send(x_new[i], nextProcess);
33:                 end if
34:             else if (i > 0)  AND  (i < processChunk − 1) then
35:                 prevValue = x[i − 1];
36:                 nxtValue = x[i + 1];
37:                 x_new[i] = (b[i] − L[i] * prevValue − U[i] * nxtValue)/D[i];
38:             end if
```

If K is the number of iterations, then the cost of the Parallel Jacobi algorithm is $O(K * N)$ for a serial execution and $O(K * N/P)$ for a parallel one with P processes. Things here are just like in the Explicit Finite-Differences scheme, since we have the same number of messages that need to be exchanged at the

Algorithm 12 Parallel Jacobi Continued

```
39:     else if processID == nofProcesses − 1 then
40:         if i == 0 then
41:             if (k == 1)  AND  (!sent) then
42:                 sent = true;
43:                 Send(b[i], previousProcess);
44:             end if
45:             Recv(prevValue, previousProcess);
46:             if processChunk > 1 then
47:                 nxtValue = x[i + 1];
48:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
49:                 if k! = iterations then
50:                     Send(x_new[i], previousProcess);
51:                 end if
52:             else
53:                 nxtValue = 0.0;
54:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
55:             end if
56:         else if i == processChunk − 1 then
57:             nxtValue = 0.0;
58:             prevValue = x[i − 1];
59:             x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
60:         else if (i > 0)  AND  (i < processChunk − 1) then
61:             prevValue = x[i − 1];
62:             nxtValue = x[i + 1];
63:             x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
64:         end if
65:     else if (processID > 0)  AND  (processID < nofProcesses − 1)
    then
66:         if (k == 1)  AND  (!sent) then
67:             sent = true;
68:             Send(b[i], previousProcess);
69:             Send(b[processChunk − 1], nextProcess);
70:         end if
71:         if i == 0 then
72:             Recv(prevValue, previousProcess);
73:             if processChunk > 1 then
74:                 nxtValue = x[i + 1];
75:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
76:                 if k! = iterations then
77:                     Send(x_new[i], previousProcess);
78:                 end if
79:             else
80:                 Recv(nxtValue, nextProcess);
81:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
82:                 if k! = iterations then
83:                     Send(x_new[i], nextProcess);
84:                 end if
85:             end if
```

Algorithm 13 Parallel Jacobi Continued

```
86:             else if i == processChunk − 1 then
87:                 prevValue = x[i − 1];
88:                 Recv(nxtValue, nextProcess;
89:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
90:                 if k! = iterations then
91:                     Send(x_new[i], nextProcess);
92:                 end if
93:             else if (i > 0)  AND  (i < processChunk − 1) then
94:                 prevValue = x[i − 1];
95:                 nxtValue = x[i + 1];
96:                 x_new[i] = (b[i] − L[i] ∗ prevValue − U[i] ∗ nxtValue)/D[i];
97:             end if
98:         end if
99:     end for
100:    for i = 0 to processChunk do
101:        x[i] = x_new[i];
102:    end for
103: end for
```

same cost. So, we expect the respective results for the methods that use Parallel Jacobi to solve their systems, that is the Implicit Finite-Differences method and the Crank-Nicolson method.

Algorithm 14 Parallel Implicit Finite-Differences

```
1: For each process:
2: for n = 0 to processChunk do
3:     k = processID ∗ processChunk + n;
4:     price[n] = payoff(k ∗ δx);
5:     derivativePrice[n] = price[n];
6: end for
7: for m = M − 1 to 0 do
8:     ParallelJacobi(STDVals, CoefficientsMatrix, processID, nofProcesses, processChunk);
9: end for
10: for i = 0 to processChunk do
11:     price[i] = derivativePrice[i];
12: end for
```

The Parallel Implicit Finite-Differences algorithm has a cost of $O(M)$ for each process either serial or parallel, but this is a cost on the Jacobi method. Given that the Jacobi has a rather fast convergence and that there are no restriction in the choice of α, the Implicit Finite-Differences scheme is expected to be somewhat faster than the Explicit Finite-Differences scheme. The results in Chapter 5 prove

this assumption.

3.3.1.1.3 Crank-Nicolson Method

Using the Crank-Nicolson scheme, equation (3.22) becomes:

$$\frac{V_n^{m+1}-V_n^m}{\delta t}+\frac{1}{2}\Big(\begin{array}{ll} rS_n & \frac{V_{n+1}^{m+1}-V_{n-1}^{m+1}}{2\delta S}+\frac{1}{2}\sigma^2 S_n^2\frac{V_{n+1}^{m+1}-2V_n^{m+1}+V_{n-1}^{m+1}}{\delta S^2}-rV_n^{m+1}+ \\ rS_n & \frac{V_{n+1}^{m}-V_{n-1}^{m}}{2\delta S}+\frac{1}{2}\sigma^2 S_n^2\frac{V_{n+1}^{m}-2V_n^{m}+V_{n-1}^{m}}{\delta S^2}-rV_n^{m}\Big) \end{array} \quad (3.63)$$

After working on the equation above, we end up to the following equation:

$$A_n^{(L)}V_{n-1}^{m+1}+B_n^{(L)}V_n^{m+1}+C_n^{(L)}V_{n+1}^{m+1}=A_n^{(R)}V_{n-1}^{m}+B_n^{(R)}V_n^{m}+C_n^{(R)}V_{n+1}^{m} \quad (3.64)$$

where

$$A_n^{(L)}=\frac{1}{4}\left(n^2\sigma^2-nr\right)\delta t, \quad (3.65)$$

$$B_n^{(L)}=1-\frac{1}{2}\left(n^2\sigma^2+r\right)\delta t, \quad (3.66)$$

$$C_n^{(L)}=\frac{1}{4}\left(n^2\sigma^2+nr\right)\delta t \quad (3.67)$$

and

$$A_n^{(R)}=\frac{1}{4}\left(nr-n^2\sigma^2\right)\delta t, \quad (3.68)$$

$$B_n^{(R)}=1+\frac{1}{2}\left(n^2\sigma^2+r\right)\delta t, \quad (3.69)$$

$$C_n^{(R)}=-\frac{1}{4}\left(nr+n^2\sigma^2\right)\delta t \quad (3.70)$$

Now, we have to adjust the boundary conditions, in order to solve the problem numerically. For a european call option, we have the following boundary conditions:

- At expiry, where $T=M\delta t$, we know that the option value equals the payoff:

$$V_n^M=\max(n\delta S-E,0) \quad (3.71)$$

- At $S=0$, we have that $n=0$ and, so

$$A_0^{(L)}=0 \quad B_0^{(L)}=1-\frac{1}{2}r\delta t \quad C_0^{(L)}=0 \quad (3.72)$$

and

$$A_0^{(R)}=0 \quad B_0^{(R)}=1+\frac{1}{2}r\delta t \quad C_0^{(R)}=0 \quad (3.73)$$

- At $S = S^*$, we have that $n = N$ and the equation becomes:

$$A_N^{(L)} V_{N-1}^{m+1} + B_N^{(L)} V_N^{m+1} + C_N^{(L)} V_{N+1}^{m+1} = A_N^{(R)} V_{N-1}^m + B_N^{(R)} V_N^m + C_N^{(R)} V_{N+1}^m \tag{3.74}$$

Checking back $C_N^{(L)}$ and $C_N^{(R)}$, we can see that $C_N^{(L)} = -C_N^{(R)}$. So, we can rearrange the equation above as follows:

$$A_N^{(L)} V_{N-1}^{m+1} + B_N^{(L)} V_N^{m+1} + C_N^{(L)} \left(V_{N+1}^{m+1} + V_{N+1}^m\right) = A_N^{(R)} V_{N-1}^m + B_N^{(R)} V_N^m \tag{3.75}$$

Using the condition

$$\lim_{S \to \infty} \frac{\partial^2 V}{\partial S^2} = 0 \tag{3.76}$$

for the Crank-Nicolson method, we have that:

$$V_{N+1}^{m+1} - 2V_N^{m+1} + V_{N-1}^{m+1} + V_{N+1}^m - 2V_N^m + V_{N-1}^m = 0$$

and so,

$$V_{N+1}^{m+1} + V_{N+1}^m = 2V_N^{m+1} + 2V_N^m - V_{N-1}^{m+1} - V_{N-1}^m \tag{3.77}$$

By substituting equation (3.77) in equation (3.75), we end up to the following equation:

$$\begin{aligned} \left(A_N^{(L)} - C_N^{(L)}\right) V_{N-1}^{m+1} + \left(B_N^{(L)} + 2C_N^{(L)}\right) V_N^{m+1} = \\ \left(A_N^{(R)} - C_N^{(R)}\right) V_{N-1}^m + \left(B_N^{(R)} + 2C_N^{(R)}\right) V_N^m \end{aligned} \tag{3.78}$$

Let,

$$\hat{A}_N^{(L)} = A_N^{(L)} - C_N^{(L)} \quad and \quad \hat{B}_N^{(L)} = B_N^{(L)} + 2C_N^{(L)} \tag{3.79}$$

and

$$\hat{A}_N^{(R)} = A_N^{(R)} - C_N^{(R)} \quad and \quad \hat{B}_N^{(R)} = B_N^{(R)} + 2C_N^{(R)} \tag{3.80}$$

So, the final boundary condition is:

$$\hat{A}_N^{(L)} V_{N-1}^{m+1} + \hat{B}_N^{(L)} V_N^{m+1} = \hat{A}_N^{(R)} V_{N-1}^m + \hat{B}_N^{(R)} V_N^m \tag{3.81}$$

where

$$\hat{A}_N^{(L)} = -\frac{1}{2} nr\delta t \quad and \quad \hat{B}_N^{(L)} = 1 + \frac{1}{2}(n-1)r\delta t \tag{3.82}$$

and

$$\hat{A}_N^{(R)} = \frac{1}{2} nr\delta t \quad and \quad \hat{B}_N^{(R)} = 1 + \frac{1}{2}(r - nr)\delta t \tag{3.83}$$

Now, that we have set up both the equations and the boundary conditions, we need to solve the system of equations that is being formed. The system consists of N linear equations. Bearing in mind that we want to parallelize the solution of this problem, let's say by creating p processes, it would better be $N = \alpha p$ since we would like each process to do the same computations with all the other ones. Next, we will give an example illustrating how we solve this system of equations.

Consider four processes, $x = 4$, and eight discretization points, $N = 8$. So, we have to solve the following linear system:

$$\begin{cases} B_0^{(L)} V_0^{m+1} = B_0^{(R)} V_0^m \\ A_1^{(L)} V_0^{m+1} + B_1^{(L)} V_1^{m+1} + C_1^{(L)} V_2^{m+1} = A_1^{(R)} V_0^m + B_1^{(R)} V_1^m + C_1^{(R)} V_2^m \\ A_2^{(L)} V_1^{m+1} + B_2^{(L)} V_2^{m+1} + C_2^{(L)} V_3^{m+1} = A_2^{(R)} V_1^m + B_2^{(R)} V_2^m + C_2^{(R)} V_3^m \\ A_3^{(L)} V_2^{m+1} + B_3^{(L)} V_3^{m+1} + C_3^{(L)} V_4^{m+1} = A_3^{(R)} V_2^m + B_3^{(R)} V_3^m + C_3^{(R)} V_4^m \\ A_4^{(L)} V_3^{m+1} + B_4^{(L)} V_4^{m+1} + C_4^{(L)} V_5^{m+1} = A_4^{(R)} V_3^m + B_4^{(R)} V_4^m + C_4^{(R)} V_5^m \\ A_5^{(L)} V_4^{m+1} + B_5^{(L)} V_5^{m+1} + C_5^{(L)} V_6^{m+1} = A_5^{(R)} V_4^m + B_5^{(R)} V_5^m + C_5^{(R)} V_6^m \\ A_6^{(L)} V_5^{m+1} + B_6^{(L)} V_6^{m+1} + C_6^{(L)} V_7^{m+1} = A_6^{(R)} V_5^m + B_6^{(R)} V_6^m + C_6^{(R)} V_7^m \\ \hat{A}_7^{(L)} V_6^{m+1} + \hat{B}_7^{(L)} V_7^{m+1} = \hat{A}_7^{(R)} V_6^m + \hat{B}_7^{(R)} V_7^m \end{cases} \tag{3.84}$$

Let us denote that the first and the last equation have been imposed by the corresponding boundary conditions.

As we can observe, the left side of the system (3.84) is known. We begin at expiry, where the option price is known to be equal to the payoff, and then we move backwards. If we replace the left side of equation with the corresponding value, we can easily see that the Crank-Nicolson scheme reduces to the Implicit Finite-Differences Scheme, as shown in (3.3.1.1.2).

The computations are more complex here, than in the Implicit Finite-Differences Scheme, since we have to include more values to compute the known quantity. This means much more communication between the processes, since the processes on both the left and the right side need to exchange the respective values. In comparison with the implicit finite-differences scheme, here the communication cost is doubled. The processes communicate using the Message Passing Interface (MPI). For more details, please see the source code.

Next, we give the algorithm for European options pricing using the Crank-Nicolson scheme.

Algorithm 15 Parallel Crank-Nicolson

1: **For each process:**
2: **for** $n = 0$ to $processChunk$ **do**
3: $k = processID * processChunk + n$;
4: $CalculatedPrice[n] = payoff(k * \delta S)$;
5: $derivativePrice[n] = CalculatedPrice[n]$;
6: **end for**
7: **for** $m = M - 1$ to 0 **do**
8: $ParallelJacobi(STDVals, CoefficientsMatrix, processID, nofProcesses, processChunk)$;
9: **end for**
10: **for** $i = 0$ to $processChunk$ **do**
11: $CalculatedPrice[i] = derivativePrice[i]$;
12: **end for**

The Parallel Crank-Nicolson algorithm has the same cost as the Parallel Implicit Finite-Differences algorithm, since they both have a loop where they call the Jacobi algorithm. Crank-Nicolson converges faster [15, 14], though, so we expect a faster execution than the Implicit Finite-Differences scheme. Please refer to Chapter 5 to see the experimental results.

3.3.2 American Options

The use of Partial Differential Equations and especially of the Finite-Difference scheme to solve them is more tricky for American options. That is because the possibility of early exercise makes them a free boundaries problem and, so, we do not know where they are in order to impose them and solve the equation.

According to [15], there are two strategies to solve free boundaries problems using Finite-Differences. The first is to track the free boundary problem as a time-stepping process. But, in American options evaluation, both the boundary conditions are implicit and, so, they do not give a direct expression for the free boundary or its time derivatives. The second is to transform the problem to one with fixed boundary conditions, solve it, and then go back to the initial problem.

We will use the diffusion equation and the Crank-Nicolson scheme to solve this problem. Of course, the right for early exercise will change the set of equations to:

$$\left(\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2}\right) \geq 0 \tag{3.85}$$

$$(u(x,\tau) - g(x,\tau)) \geq 0 \tag{3.86}$$

$$\left(\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2}\right)(u(x,\tau) - g(x,\tau)) = 0 \tag{3.87}$$

where $g(x,\tau)$ is the transformed payoff constraint function and is given by:

$$g(x,\tau) = e^{\frac{1}{4}(k+1)^2\tau} \max\left(e^{\frac{1}{2}(k-1)x} - e^{\frac{1}{2}(k+1)x}, 0\right) \tag{3.88}$$

for the put and

$$g(x,\tau) = e^{\frac{1}{4}(k+1)^2\tau} \max\left(e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}, 0\right) \tag{3.89}$$

for the call. The initial boundary conditions now become:

$$\begin{aligned} u(x,0) &= g(x,0), \\ u(x,\tau)&\text{iscontinuous}, \\ \frac{\partial u}{\partial x}(x,\tau)&\text{is as continuous as} g(x,\tau), \\ \lim_{x\to\infty} u(x,\tau) &= \lim_{x\to\infty} g(x,\tau) \end{aligned} \tag{3.90}$$

So, if we solve these equations, then we can find the free boundary $x = x_f(\tau)$ a posteriori by the condition that defines it, that is:

$$u(x_f(\tau),\tau) = g(x_f(\tau),\tau), \quad \text{but} \quad u(x,\tau) > g(x,\tau) \text{ for } x > x_f(\tau)$$

for the put and

$$u(x_f(\tau),\tau) = g(x_f(\tau),\tau), \quad \text{but} \quad u(x,\tau) > g(x,\tau) \text{ for } x < x_f(\tau)$$

for the call.

The diffusion equation, as we have transformed it for the American options, can be approximated by

$$\begin{aligned} u_n^{m+1} - \frac{1}{2}\alpha\left(u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}\right) \\ \geq u_n^m + \frac{1}{2}\alpha\left(u_{n+1}^m - 2u_n^m + u_{n-1}^m\right) \end{aligned} \tag{3.91}$$

where

$$\alpha = \frac{\delta t}{(\delta x)^2}$$

Now let's consider the descritised payoff function:

$$g_n^m = g(n\delta x, m\delta t) \tag{3.92}$$

So, we have the condition:

$$u_n^m \geq g_n^m \quad \text{for } m \geq 1 \tag{3.93}$$

and the boundary and initial conditions:

$$u_{N^-}^m = g_{N^-}^m, \quad u_{N^+}^m = g_{N^+}^m, \quad u_n^0 = g_n^0 \tag{3.94}$$

In the same spirit, Equation (3.10) becomes:

$$(1+\alpha)u_n^{m+1} - \frac{1}{2}\alpha(u_{n-1}^{m+1} + u_{n+1}^{m+1}) \geq Z_n^m \tag{3.95}$$

So, now, Equation (3.87) is approximated by:

$$\left((1+\alpha)u_n^{m+1} - \frac{1}{2}\alpha(u_{n-1}^{m+1} + u_{n+1}^{m+1}) - Z_n^m\right)\left(u_n^{m+1} - g_n^{m+1}\right) = 0 \tag{3.96}$$

From this point on, we follow the same procedure as in Section 3.2.1.3 and we have:

$$u^m = \begin{pmatrix} u_{N^-+1}^m \\ \vdots \\ u_{N^+-1}^m \end{pmatrix}, \quad g^m = \begin{pmatrix} g_{N^-+1}^m \\ \vdots \\ g_{N^+-1}^m \end{pmatrix} \tag{3.97}$$

and

$$b^m = \begin{pmatrix} b_{N^-+1}^m \\ \vdots \\ b_0^m \\ \vdots \\ b_{N^+-1}^m \end{pmatrix} = \begin{pmatrix} Z_{N^-+1}^m \\ \vdots \\ Z_0^m \\ \vdots \\ Z_{N^+-1}^m \end{pmatrix} + \frac{1}{2}\alpha \begin{pmatrix} g_{N^-}^{m+1} \\ 0 \\ \vdots \\ 0 \\ g_{N^+}^{m+1} \end{pmatrix} \tag{3.98}$$

Now, Equation (3.11) becomes:

$$\begin{gathered} Cu^{m+1} \geq b^m, \quad u^{m+1} \geq g^{m+1} \\ \left(u^{m+1} - g^{m+1}\right)\left(Cu^{m+1} - b^m\right) = 0 \end{gathered} \tag{3.99}$$

where C is given by Equation (3.12).

In order to solve this system, we take the projected SOR method. Next we give the algorithm for solving this problem, as given in [15].

Algorithm 16 Projected SOR Method for American Options using Crank-Nicolson Scheme

1: $loops = 0$;
2: $\alpha_2 = \alpha/2.0$;
3: $u[N^-] = g[N^-]$;
4: $u[N^+] = g[N^+]$;
5: **repeat**
6: $error = 0.0$;
7: **for** $n = N^- + 1$ to N^+ **do**
8: $y = (b[n] + \alpha_2 * (u[n-1] + u[n+1]))/(1+\alpha)$;
9: $y = \max(g[n], u[n] + \omega * (y - u[n]))$;
10: $error = error + (u[n] - y) * (u[n] - y)$;
11: $u[n] = y$;
12: **end for**
13: $++loops$;
14: **until** $error > eps$;
return loops

Chapter 4

Binomial Trees

4.1 Introduction

In finance, the binomial options pricing model provides a numerical method for the valuation of options. The binomial model was first proposed by *Cox, Ross* and *Rubinstein* [11]. Essentially, this model uses a discrete-time model of the varying price over time of the underlying financial instrument and it arises from discrete random walk models of the underlying security.

The binomial methods are based on two main assumptions. The first one is that the continuous asset random walk can be modelled by a discrete random walk with the following properties:

- The asset price S changes only from one step to another, meaning from $m\delta t$ to $(m+1)\delta t$ until the expiration date $T = M\delta t$. We denote that δt is a small but not infinitesimal time-step between movements in the asset price.
- Considering an asset price S^m at time-step $m\delta t$, then at time $(m+1)\delta t$ the asset price S^{m+1} should be either $uS^m > S^m$ or $dS^m < S^m$. This means that the asset price may go up multiplied by u or may go down multiplied by d. Obviously, it should be $u > 1$ and $d < 1$.
- The probability p of S going up to uS is known (as is the probability (1-p) of S going down to dS).

The second assumption is that of a risk-neutral world (explained in Appendix A), that is, one where an investor's risk preferences are irrelevant to derivative security valuation. This assumption may be made whenever it is possible to hedge a portfolio perfectly and make it riskless.

The main idea of this method is to divide the time to expiration in discrete time slots (δt) and during each slot $(m+1)\delta t$ we consider that the underlying asset S^{m+1} can take two possible values: uS^m or dS^m. In this context, we are building a tree including all possible scenarios for the asset moving up or down during each time slot. When we have built the whole tree, we know the values of the last nodes, as they are the option prices at expiry and can be easily computed, and then we are moving backwards computing the value of every node, that is, the asset and, of course, the option price at each slot.

4.2 Method Overview

As we already know, the asset price random walk can be modelled by the stochastic differential equation:

$$\frac{dS}{S} = \sigma dX + \mu dt. \tag{4.1}$$

We recall our second assumption of a risk-neutral world. This means that we can replace equation (4.1) with the following equation:

$$\frac{dS}{S} = \sigma dX + r dt. \tag{4.2}$$

In this model, we assume that every δt the asset goes up by u or down by d. So, if we are at the slot $m\delta t$ and we have calculated the option price for the slot $(m+1)\delta t$ then we can get the option price for slot $m\delta t$ by taking the expected value of V^{m+1} discounted by the risk-free interest rate r:

$$V^m = E[e^{-r\delta t} V^{m+1}] \tag{4.3}$$

Note that first we are calculating the binomial tree for the asset price, then calculate the option price at expiry (i.e. the payoff) and then we go back using equation (4.3) to value the option at each time slot.

A binomial tree can look like the one shown in Figure 4.2

4.3 Parameters Computations

In this Section we will see how the parameters u, d, p used for computations by the binomial model are computed. The main idea is that the discrete random walk illustrated in Figure 4.2 and the continuous random walk of equation (4.1) should have the same mean and variance. So, if at time step $m\delta t$ the asset price is S^m, we equate the expected values and variances of S^{m+1} under the continuous risk-neutral random walk (4.1) and the discrete binomial model 4.2.

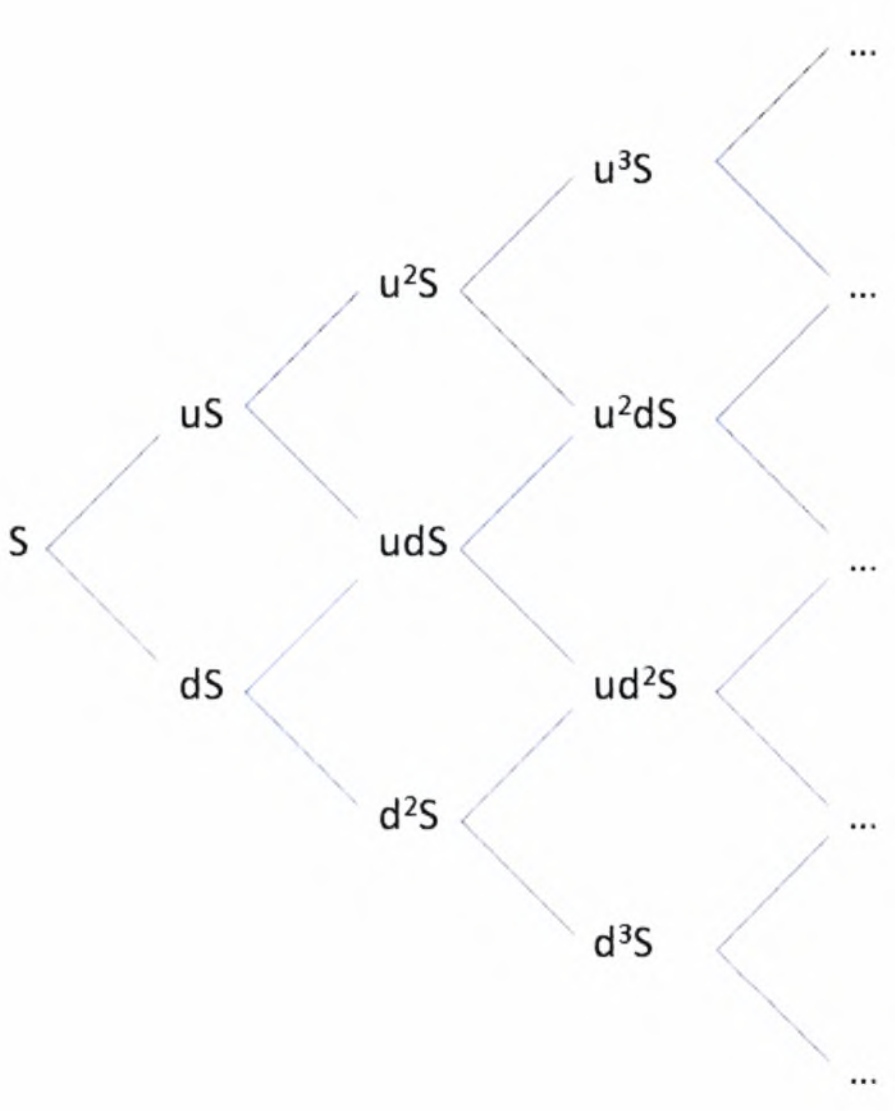


Figure 4.1: Binomial Tree

Firstly, we are going to calculate the expected value of the continuous and the discrete binomial random walk model. Suppose that at time $m\delta t$, the asset value is S^m. Then, the expected value of S^{m+1} for the continuous random walk will be:

$$E_c[S^{m+1}|S^m] = e^{r\delta t}S^m \tag{4.4}$$

and the expected value of S^{m+1} for the discrete binomial random walk will be:

$$E_b[S^{m+1}|S^m] = (pu + (1-p)\,d)\,S^m \tag{4.5}$$

Equating these two expected values, we have that:

$$pu + (1-p)\,d = e^{r\delta t}. \tag{4.6}$$

Secondly, we are going to calculate the variance of the continuous and the discrete binomial random walk model. Again, let's suppose that at time $m\delta t$, the asset value is S^m. Then, the variance of S^{m+1}, given S^m, for the continuous random walk will be:

$$var_c[S^{m+1}|S^m] = e^{2r\delta t}\left(e^{\sigma^2\delta t - 1}\right)(S^m)^2 \tag{4.7}$$

and the variance of S^{m+1}, given S^m, for the discrete binomial random walk will be:

$$var_b[S^{m+1}|S^m] = \left(pu^2 + (1-p)\,d^2 - e^{2r\delta t}\right)(S^m)^2 \tag{4.8}$$

Equating these two variances, we have that:

$$pu^2 + (1-p)\,d^2 = e^{(2r+\sigma^2)\delta t} \tag{4.9}$$

So, we have two equations, (4.6) and (4.9), for three unknowns, u, d, and p. In order to define all these three unknowns uniquely, we need one more equation. Unfortunately there is no theoretic background to lead us to another equation. Our choice for the final one will somewhat arbitrary. The most popular choices are:

$$u = \frac{1}{d} \tag{4.10}$$

and

$$p = \frac{1}{2} \tag{4.11}$$

Next, we are demonstrating the final formula for each one of these cases.

4.3.1 The Case u = 1/d

In this case, we have equations (4.6), (4.9) and (4.10) to define our three unknowns. For these three equations, we have that:

$$d = A - \sqrt{A^2 - 1}, \tag{4.12}$$

$$u = A + \sqrt{A^2 - 1} \tag{4.13}$$

and

$$p = \frac{e^{r\delta t} - d}{u - d} \tag{4.14}$$

where

$$A = \frac{1}{2}\left(e^{-r\delta t} + e^{(r+\sigma^2)\delta t}\right) \tag{4.15}$$

This choice leads to a tree in which the starting asset price reccurs every even time step and which is symmetric about this price. The asset price drift, caused by the $r\delta t$ term in (4.1), is reflected in the fact that the probability of an up movement differs from the probability of a down movement, since $p \neq (1 - p)$.

4.3.2 The Case p = 1/2

In this case we have equations (4.6), (4.9) and (4.11) to define our three unknowns. For these three equations, we have that:

$$d = e^{r\delta t}\left(1 - \sqrt{e^{\sigma^2\delta t} - 1}\right), \tag{4.16}$$

$$u = e^{r\delta t}\left(1 + \sqrt{e^{\sigma^2\delta t} - 1}\right) \tag{4.17}$$

and

$$p = \frac{1}{2}. \tag{4.18}$$

This choice consider equal probabilities for an up and a down movement and we find that $ud > 1$ and that the tree is oriented in the direction of the drift. If we take a very large time step, then d may become negative, in which case the binomial method will fail.

4.4 Binomial Tree Setup

We can choose any of the two cases, 4.3.1 or 4.3.2, to build the binomial tree, as shown in Figure 4.2. Assuming that we start from now that the asset price is known, S_0, we can build the tree as follows:

$$S_0^1 = dS_0^0 \qquad S_1^1 = uS_0^0,$$

$$S_0^2 = d^2S_0^0 \qquad S_1^2 = udS_0^0 \qquad S_2^2 = u^2S_0^0$$

and so on. At time step $m\delta t$ each possible value for S is given by the following formula:

$$S_n^m = d^{m-n}u^n S_0^0, \quad n = 0, 1, \ldots, m. \tag{4.19}$$

This way we can build the binomial tree for our asset and then use it to value an option on it. This method is shown in the next Section 4.5.

4.5 Option Pricing

Consider an option that we can value at expiry, that is, we can calculate its payoff, such as a call or a put option. Suppose that we have M time steps. Then, at expiry, i.e. at time step $M\delta t$, it is:

$$V_n^M = \max(K - S_n^M, 0), \quad n = 0, 1, \ldots, M, \tag{4.20}$$

where K is the exercise price and V_n^M denotes the n-th possible value of the put at time step M. Respectively, for a call option, we have that:

$$V_n^M = \max(S_n^M - K, 0), \quad n = 0, 1, \ldots, M. \tag{4.21}$$

At this point, we can find the expected value of the option at the time step prior to expiry, that is $(M-1)\delta t$, since we know that the probability of going up is p and the probability of going down is $(1-p)$. Then, using the risk-neutral argument, we can calculate the option value at this time step and then move one step backwards, until we reach today. Next, we are demonstrating how this method works for European options.

4.5.1 European Options

Consider a European option whose value V_n^{m+1} at time step $(m+1)\delta t$ is known. We are going to calculate its value V_n^m at time step $m\delta t$. As we have seen before, V_n^m equals the discounted expected value of the option at time step $m\delta t$, keeping in mind that the probability of moving up is p and the probability of moving down is $(1-p)$. Putting it all together, we have that:

$$V_n^m = e^{-r\delta t}\left(pV_{n+1}^{m+1} + (1-p)V_n^{m+1}\right), \quad n = 0, 1, \ldots, m. \tag{4.22}$$

Since we know the option value at expiry $V_n^M, n = 0, 1, \ldots, M$, we can start from the payoff function and then recursively define the values $V_n^m, n = 0, 1, \ldots, m$ for $m < M$ until we arrive to the desired option value today V_0^0.

Next, we give an example of an algorithm which computes a European option value using the binomial method:

Algorithm 17 Binomial Method

```
1: discount = e^{-rδt};
2: array[0] = S_0;
3: for m = 1 to M do
4:     for n = m to 0 do
5:         array[n] = u * array[n − 1];
6:     end for
7:     array[0] = d * array[0];
8: end for
9: for n = 0 to M do
10:    array[n] = payoff(array[n]);
11: end for
12: for m = M to 0 do
13:    for n = 0 to m do
14:        tmp = p * array[n + 1] + (1 − p) * array[n];
15:        array[n] = discount * tmp;
16:    end for
17: end for
```

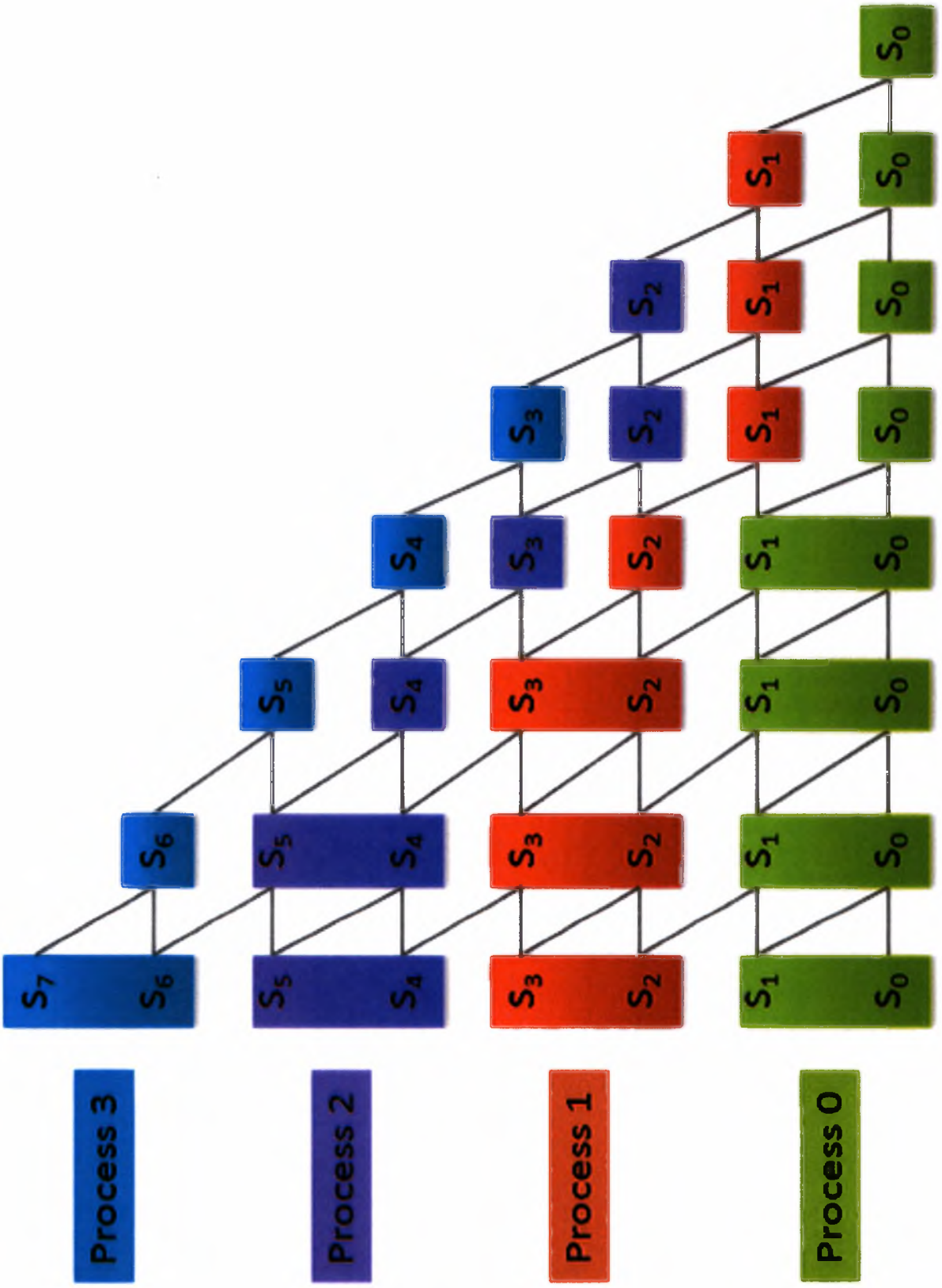
Parallel Binomial Method

The parallelization of the Binomial is done in the same way with the previous methods. The main idea is that we first setup the binomial tree which represents the expected growth of the asset and then work on that tree to price a European option. Once again, each process does the exact same thing, so that there is fairness in computational effort.

In more detail, each process builds a part of the binomial for which it has knowledge. For any other parts, the processes communicate with each other. In this thesis, the MPI framework has been used for this purpose. The parallel algorithm is much more complicated than the serial one. This happens because the tree is getting smaller and smaller; so, if one wants each process to do the same amount of computations, one needs to pay attention in the distribution of data for each process, as shown in Figure 4.2 For more details, please read the algorithm given next. Note that the algorithm assumes the tree has already been built and it works on that tree to price the option.

The cost of the Parallel Binomial algorithm is $O(M)$ for each process. It has a very good performance as the processes become more and more (until, certainly, an upper limit) since the problem that each process needs to solve becomes shorter and the cost of communication is very small: each process exchange only one message with its neighbours each time. Experimental results in Chapter 5 can prove these assumptions.

Figure 4.2: Process Allocation for Binomial Tree



Algorithm 18 Parallel Binomial Method

```
1: For each process:
2: firstPos = 0;
3: lastPos = processChunk − 1;
4: discount = e^{−r∗δt};
5: curSize = processChunk;
6: m = M;
7: for i = 0 to processChunk do
8:     assetPrice comes from the binomial tree for the asset;
9:     derivativePrice[i] = payoff(assetPrice[i]);
10:    if processID > 0 then
11:        if i == 0 then
12:            Send(derivativePrice[i], previousProcess;
13:        end if
14:    end if
15: end for
16: for m = M − 1 to 0 do
17:    tmp = mmod(nofProcesses);
18:    if processID < m then
19:        if tmp == processID then
20:            curSize − −;
21:        end if
22:    end if
23:    for i = 0 to curSize do
24:        if processID == 0 then
25:            if i == curSize − 1 then
26:                prevValue = derivativePrice[i];
27:                if tmp == processID then
28:                    nxtValue = derivativePrice[i + 1];
29:                    tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue + (1 −
   p) ∗ prevValue);
30:                else
31:                    Recv(nxtValue, nextProcess);
32:                    tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue + (1 −
   p) ∗ prevValue);
33:                    if (tmp == processID + 1)  AND  (m > 1) then
34:                        Send(tmpDerivativePrice[i], nextProcess);
35:                    end if
36:                end if
37:            else
38:                nxtValue = derivativePrice[i + 1];
39:                prevValue = derivativePrice[i];
40:                tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue + (1 − p) ∗
   prevValue);
41:            end if
```

Algorithm 19 Parallel Binomial Method Continued

```
42:        else if (processID > 0)  AND  (processID < nofProcesses − 1)
   then
43:            if processID < m then
44:                if tmp < processID then
45:                    nxtValue = derivativePrice[i];
46:                    if i == 0 then
47:                        Recv(prevValue, previousProcess);
48:                        tmpDerivativePrice[i] = discount∗(p∗nxtValue+(1−
   p) ∗ prevValue);
49:                    else
50:                        prevValue = derivativePrice[i − 1];
51:                        tmpDerivativePrice[i] = discount∗(p∗nxtValue+(1−
   p) ∗ prevValue;
52:                    end if
53:                    if (i == curSize − 1)  AND  (tmp! = 0) then
54:                        Send(tmpDerivativePrice[i], nextProcess);
55:                    else if (i == 0)  AND  (tmp == 0) then
56:                        Send(tmpDerivativePrice[i], previousProcess;
57:                    end if
58:                else
59:                    prevValue = derivativePrice[i];
60:                    if tmp > processID then
61:                        if i == curSize − 1 then
62:                            Recv(nxtValue, nextProcess);
63:                            tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue +
   (1 − p) ∗ prevValue);
64:                            if tmp == processID + 1 then
65:                                Send(tmpDerivativePrice[i], nextProcess);
66:                            end if
67:                        else
68:                            nxtValue = derivativePrice[i + 1];
69:                            tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue +
   (1 − p) ∗ prevValue);
70:                        end if
71:                        if i == 0 then
72:                            Send(tmpDerivativePrice[i], previousProcess);
73:                        end if
74:                    else if tmp == processID then
75:                        nxtValue = derivativePrice[i + 1];
76:                        tmpDerivativePrice[i] = discount∗(p∗nxtValue+(1−
   p) ∗ prevValue);
77:                        if i == curSize − 1 then
78:                            Send(tmpDerivativePrice[i], nextProcess);
79:                        end if
80:                    end if
81:                end if
82:            else
83:                break;
84:            end if
```

Algorithm 20 Parallel Binomial Method Continued

```
85:         else if processID == nofProcesses − 1 then
86:             if processID < m then
87:                 if tmp < processID then
88:                     nxtValue = derivativePrice[i];
89:                     if i == 0 then
90:                         Recv(prevValue, previousProcess);
91:                     else
92:                         prevValue = derivativePrice[i − 1];
93:                     end if
94:                     tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue + (1 −
    p) ∗ prevValue);
95:                 else if tmp == processID then
96:                     nxtValue = derivativePrice[i + 1];
97:                     prevValue = derivativePrice[i];
98:                     tmpDerivativePrice[i] = discount ∗ (p ∗ nxtValue + (1 −
    p) ∗ prevValue);
99:                 end if
100:                if tmp == 0 then
101:                    if i == 0 then
102:                        Send(tmpDerivativePrice[i]);
103:                    end if
104:                end if
105:            else
106:                break;
107:            end if
108:        end if
109:    end for
110:    for j = 0 to curSize do
111:        derivativePrice[j] = tmpDerivativePrice[j];
112:    end for
113: end for
```

At this point, let us denote that it has been helpful reading [5] for developing the source code and the parallel algorithm for the binomial method.

4.5.2 American Options

The Binomial method seems to be the most straightforward for valuing American options. That is because it is in its nature to compute the asset and the option values at various time steps. The tricky part in this method is to decide which of the two choices (exercise the option or retain it) is the best. Next, we give the algorithm from pricing American options using the Binomial method, as illustrated in [15].

Algorithm 21 Binomial Method for American Options

1: $discount = e^{-r\delta t}$;
2: $s[0][0] = S_0$;
3: **for** $m = 1$ to M **do**
4: **for** $n = m + 1$ to 0 **do**
5: $s[m][n] = u * s[m-1][n-1]$;
6: **end for**
7: $s[m][0] = d * s[m-1][0]$;
8: **end for**
9: **for** $n = 0$ to M **do**
10: $v[M][n] = payoff(s[M][n])$;
11: **end for**
12: **for** $m = M$ to 0 **do**
13: **for** $n = 0$ to m **do**
14: $hold = (1-p) * v[m+1][n] + p * v[m+1][n+1]$;
15: $hold* = discount$;
16: $v[m][n] = \max(hold, payoff(s[m][n]))$;
17: **end for**
18: **end for**

Chapter 5

Experiments

In this section, we give some results that came from a simulation ran for the parallel computational methods illustrated in the previous section. The characteristics of the options are:

- European Options
- Asset Price: 15
- Exercise Price: 10
- Constant Interest Rate: 5%
- Volatility: 20%
- Expiration: After 6 months
- Loops:
  - Monte Carlo: 100,000,000
  - Partial Differential Equations: 10,000,000
  - Binomial: 100,000

This simulation was ran on a pc with the following characteristics:

- Inter Core 2 Duo T8100 2.1 GHz CPU
- 3GB RAM
- OS: openSUSE 10.3 with Kernel 2.6.22.5-31-default
- LAM/MPI for x86-64 architecture
- GNU C++ Compiler

Figure 5.1: Monte Carlo Method - Time

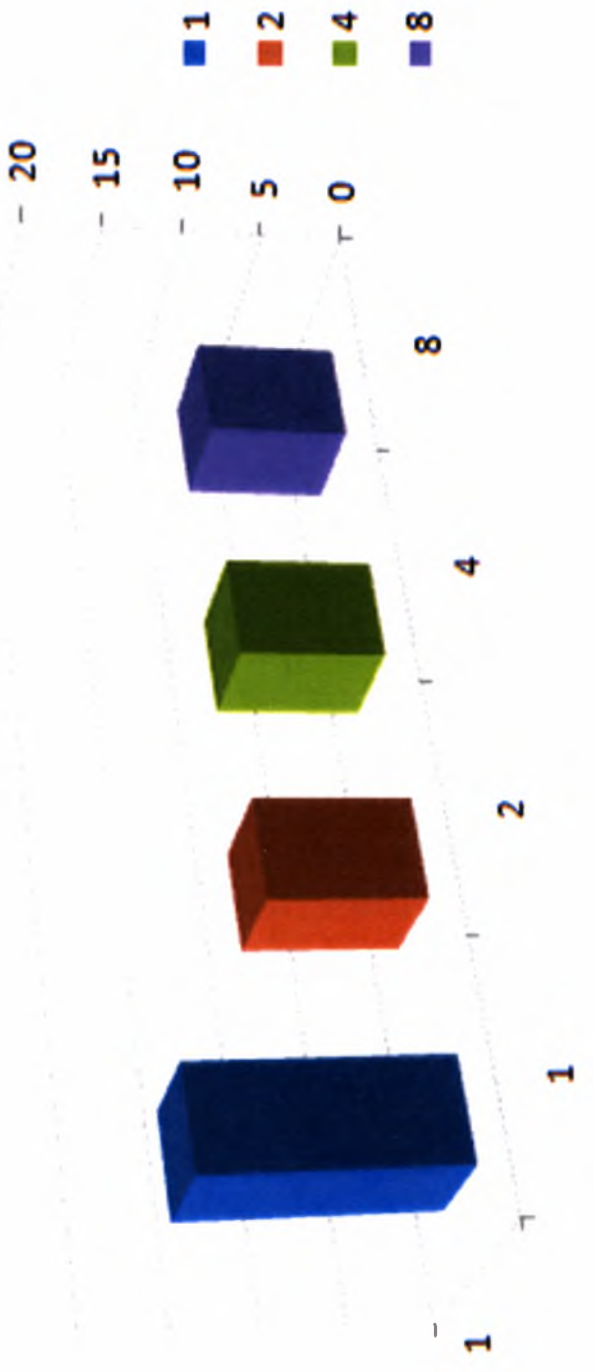


Figure 5.2: Monte Carlo Method - Memory

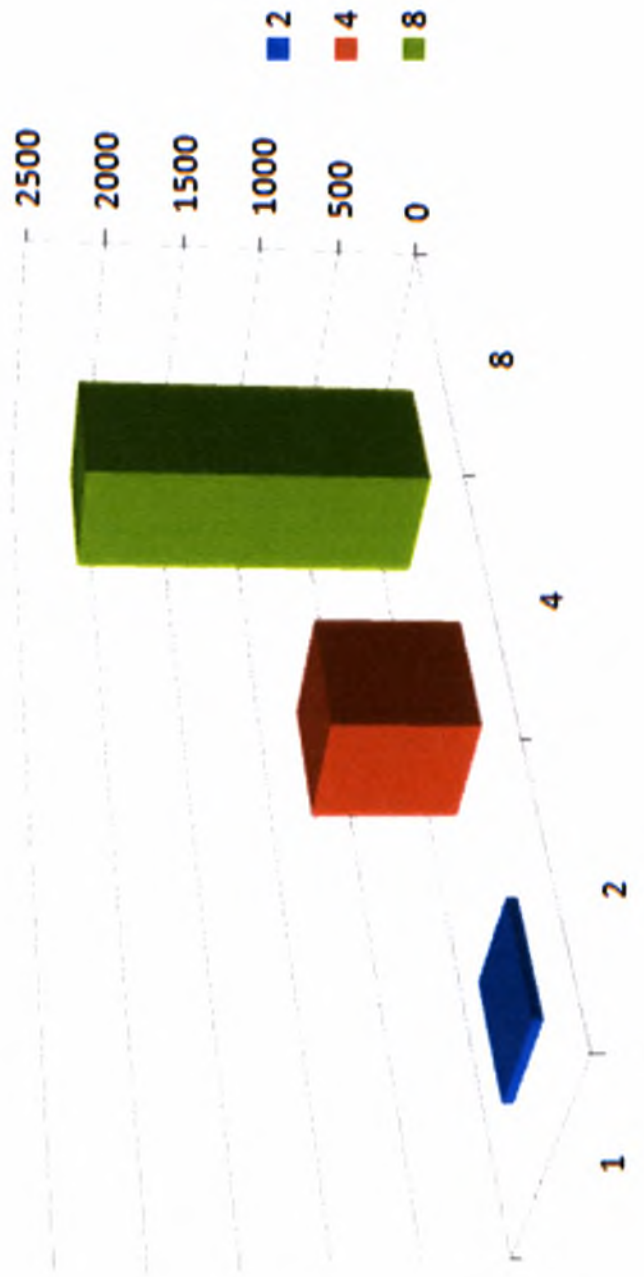


Figure 5.3: Explicit Finite-Differences Method - Time

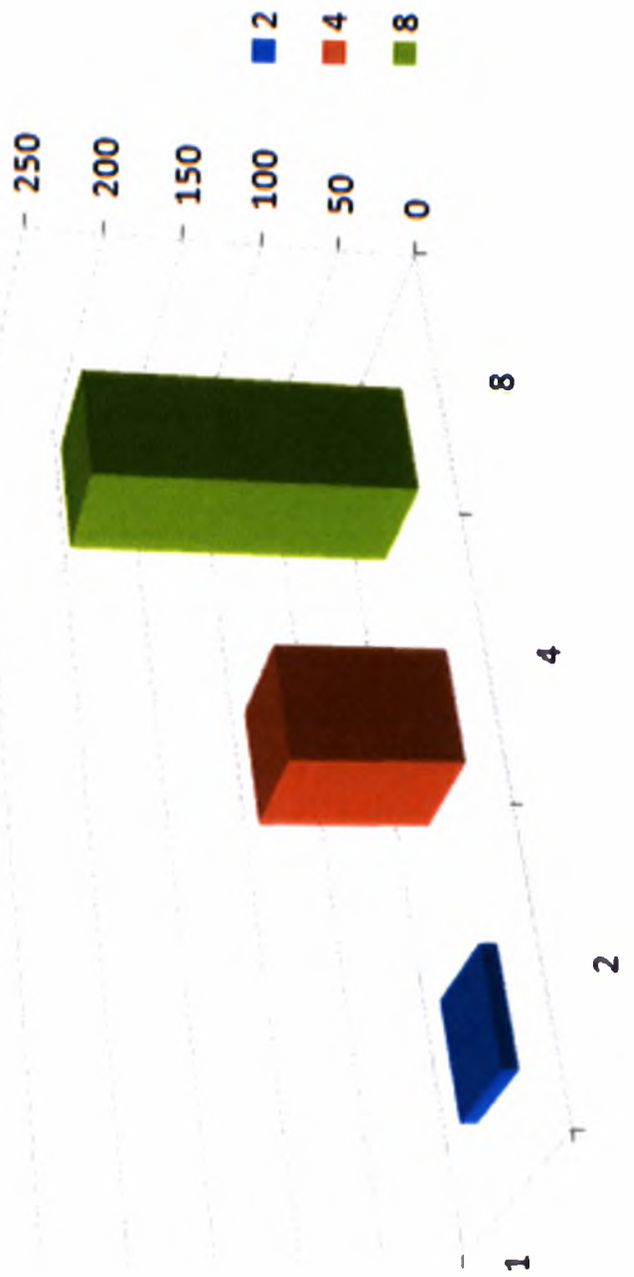


Figure 5.4: Explicit Finite-Differences Method - Memory

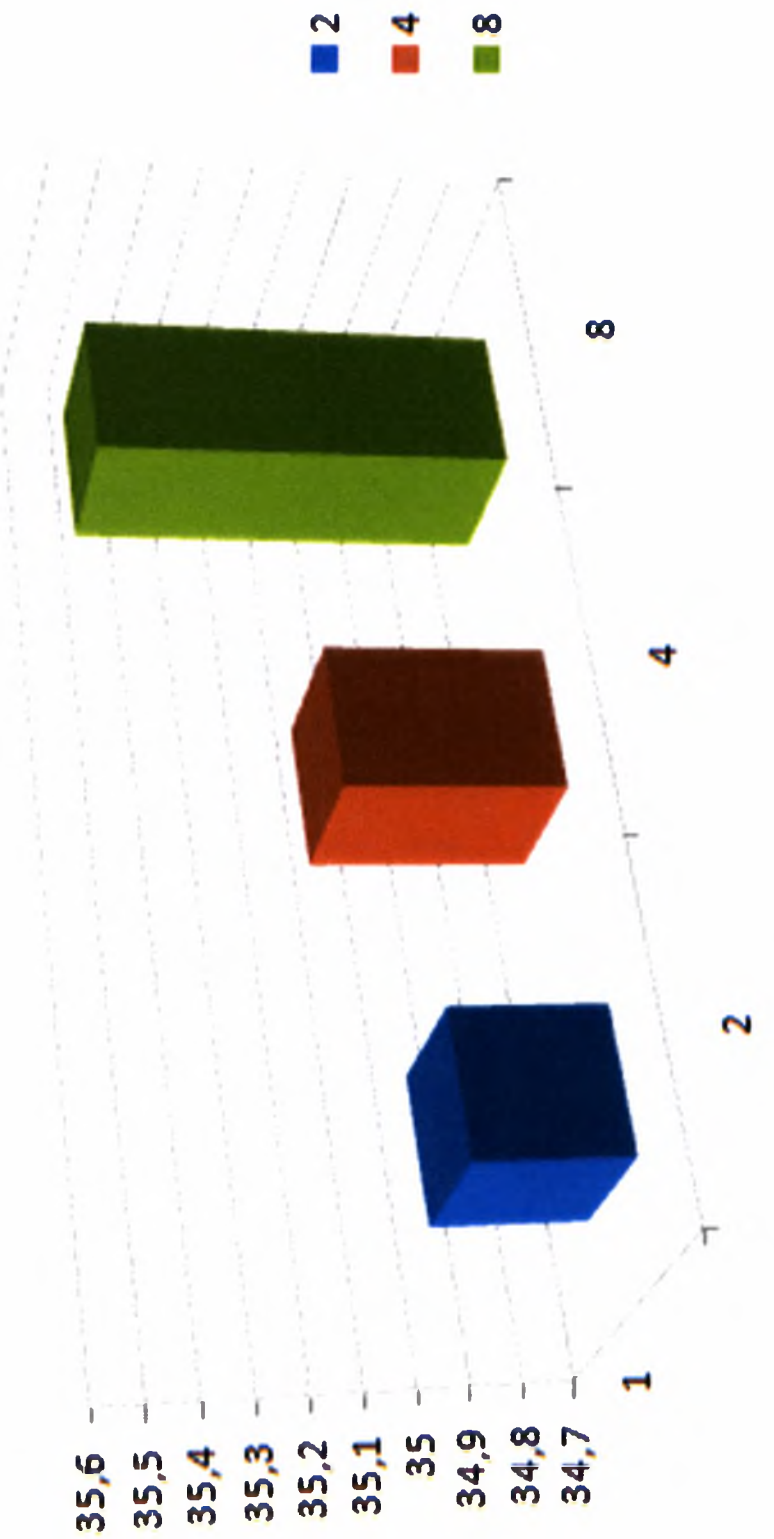


Figure 5.5: Implicit Finite-Differences Method- Time

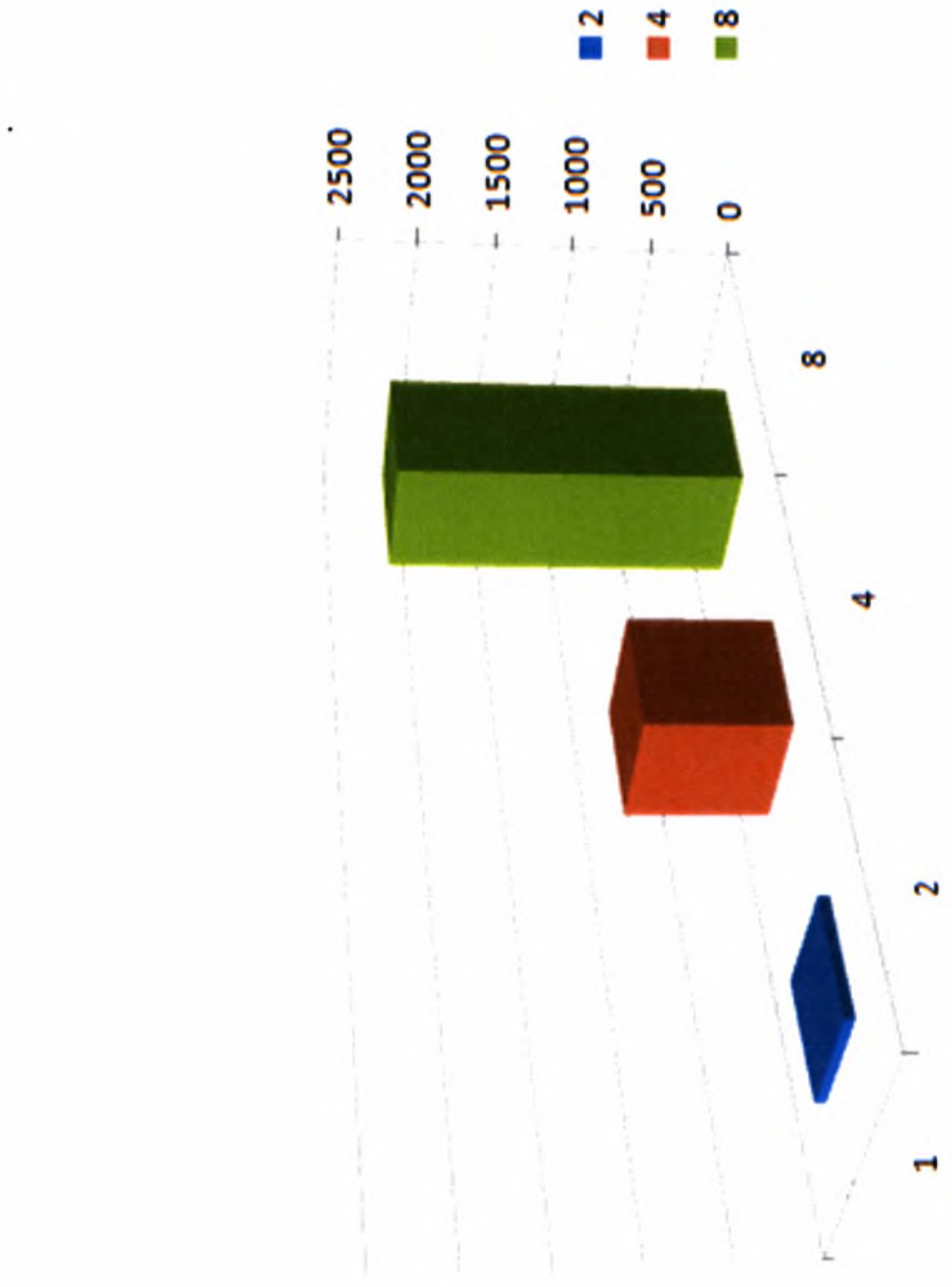


Figure 5.6: Implicit Finite-Differences Method - Memory

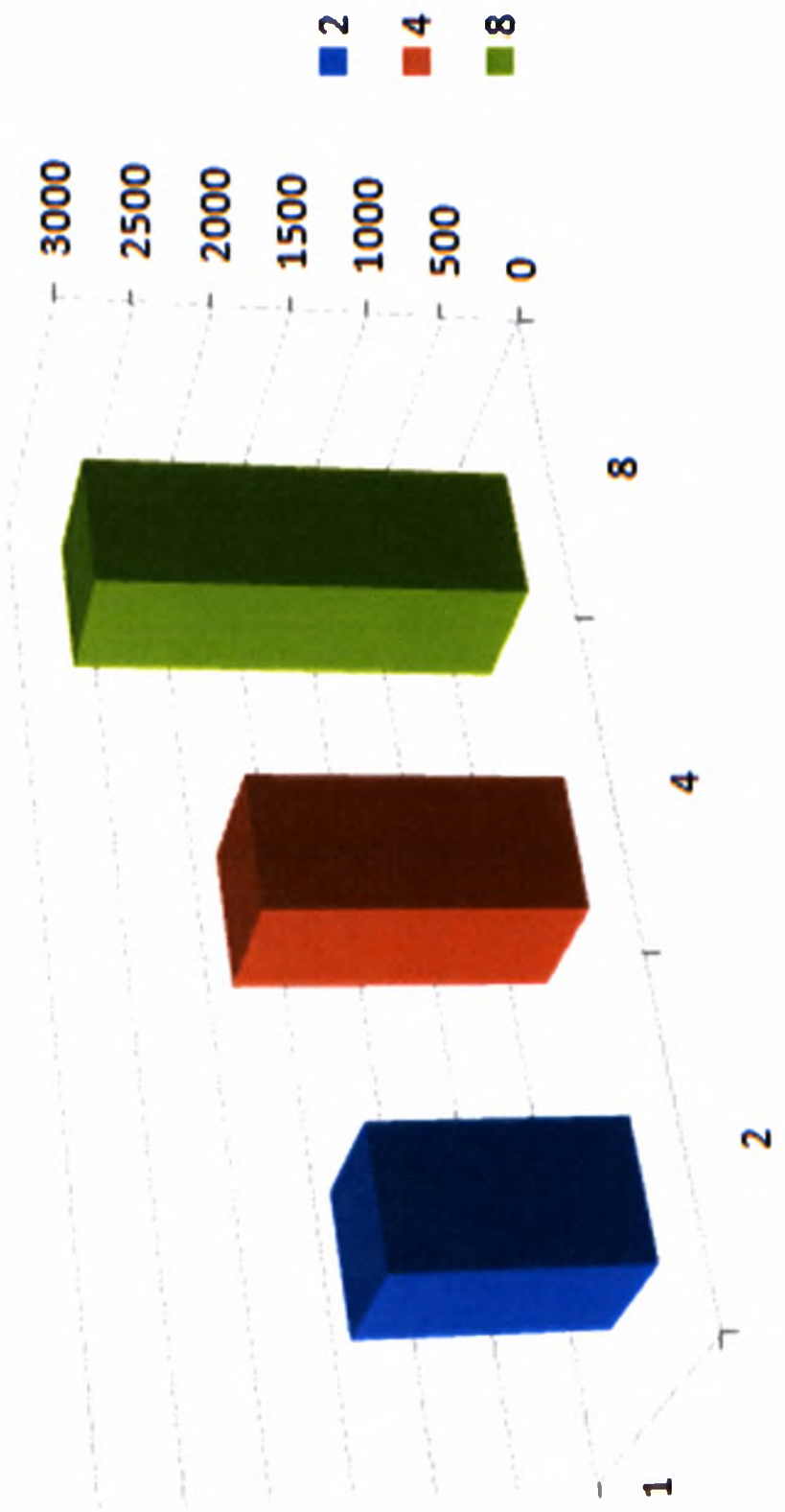


Figure 5.7: Crank-Nicolson Method - Time

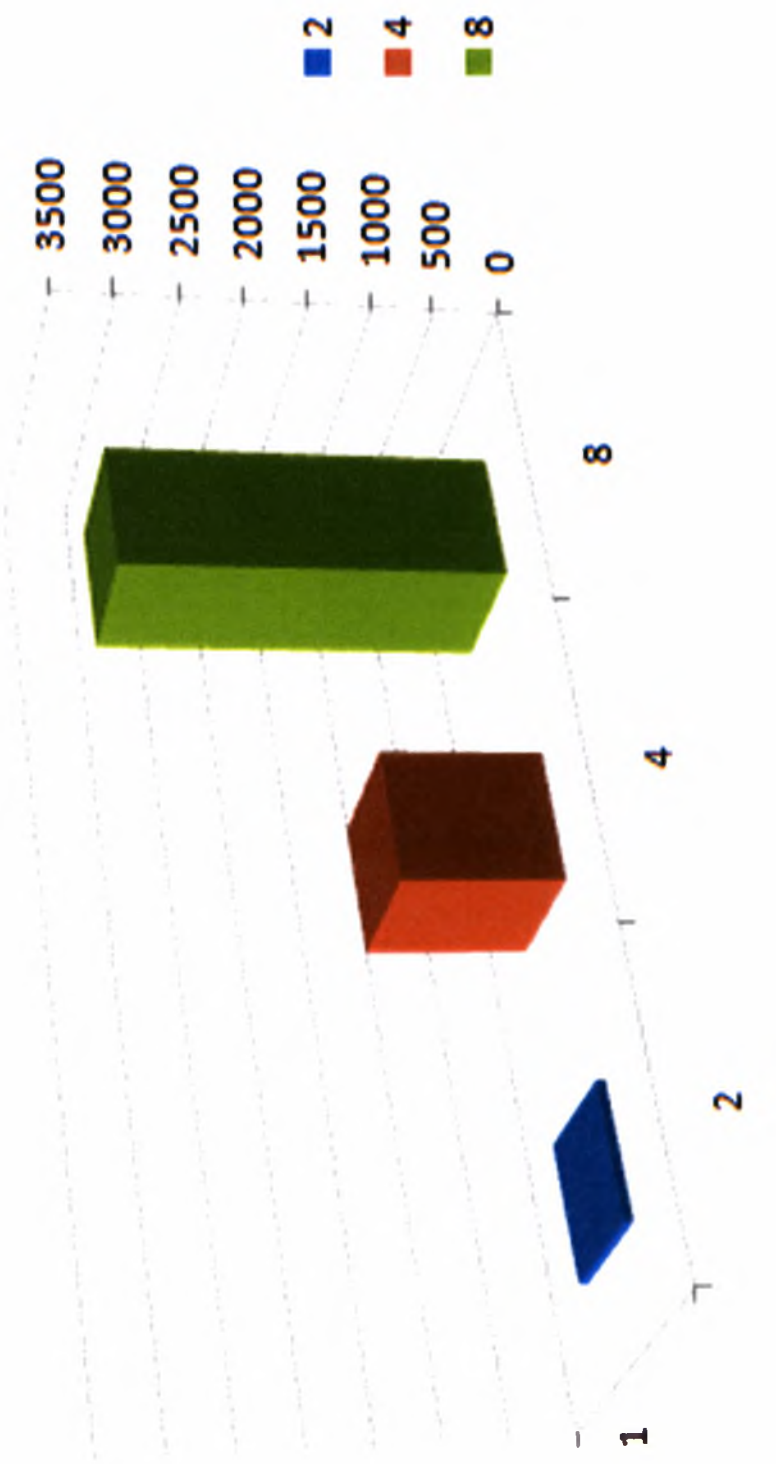


Figure 5.8: Crank-Nicolson Method - Memory

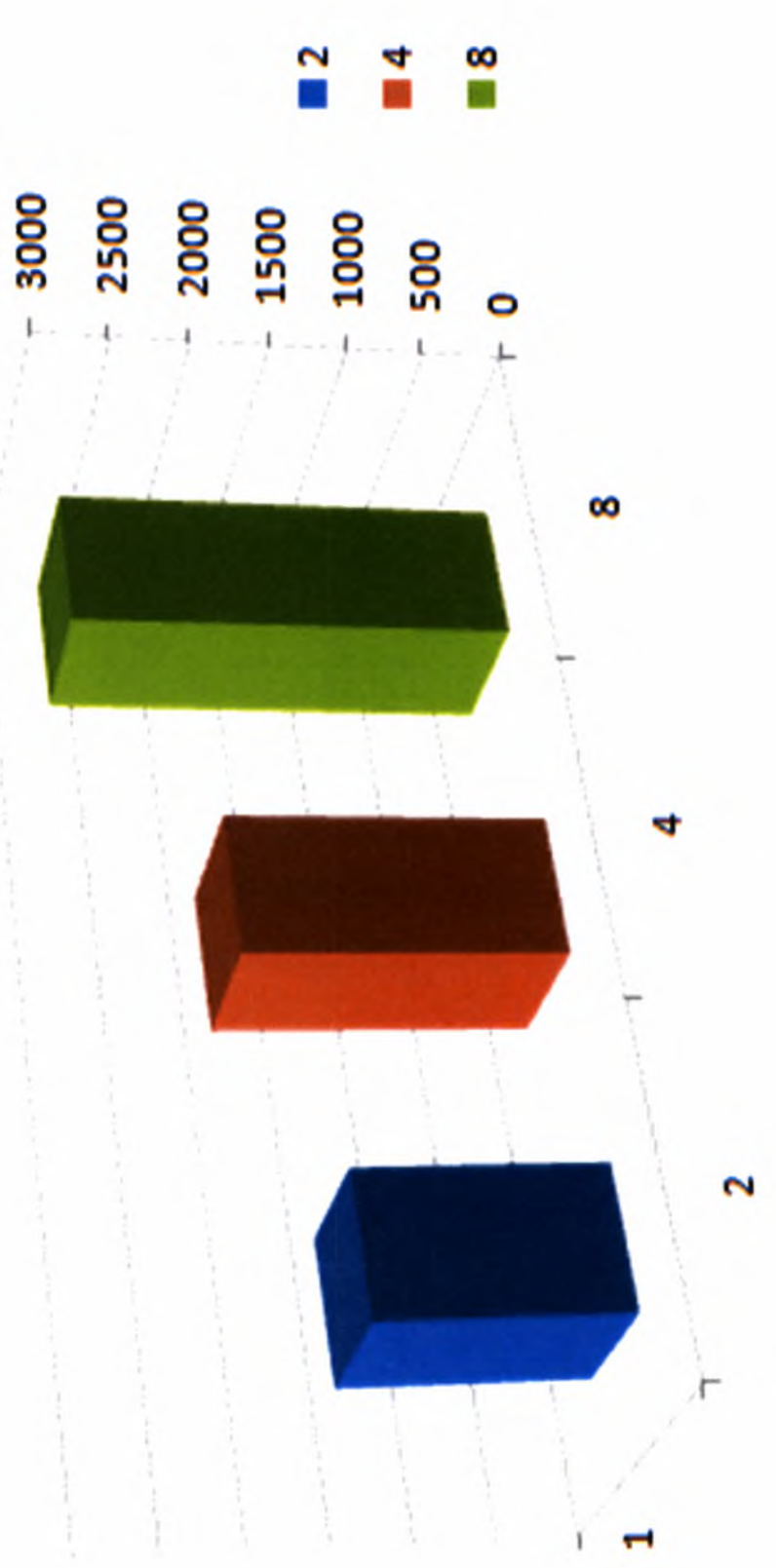


Figure 5.9: Binomial Method - Time

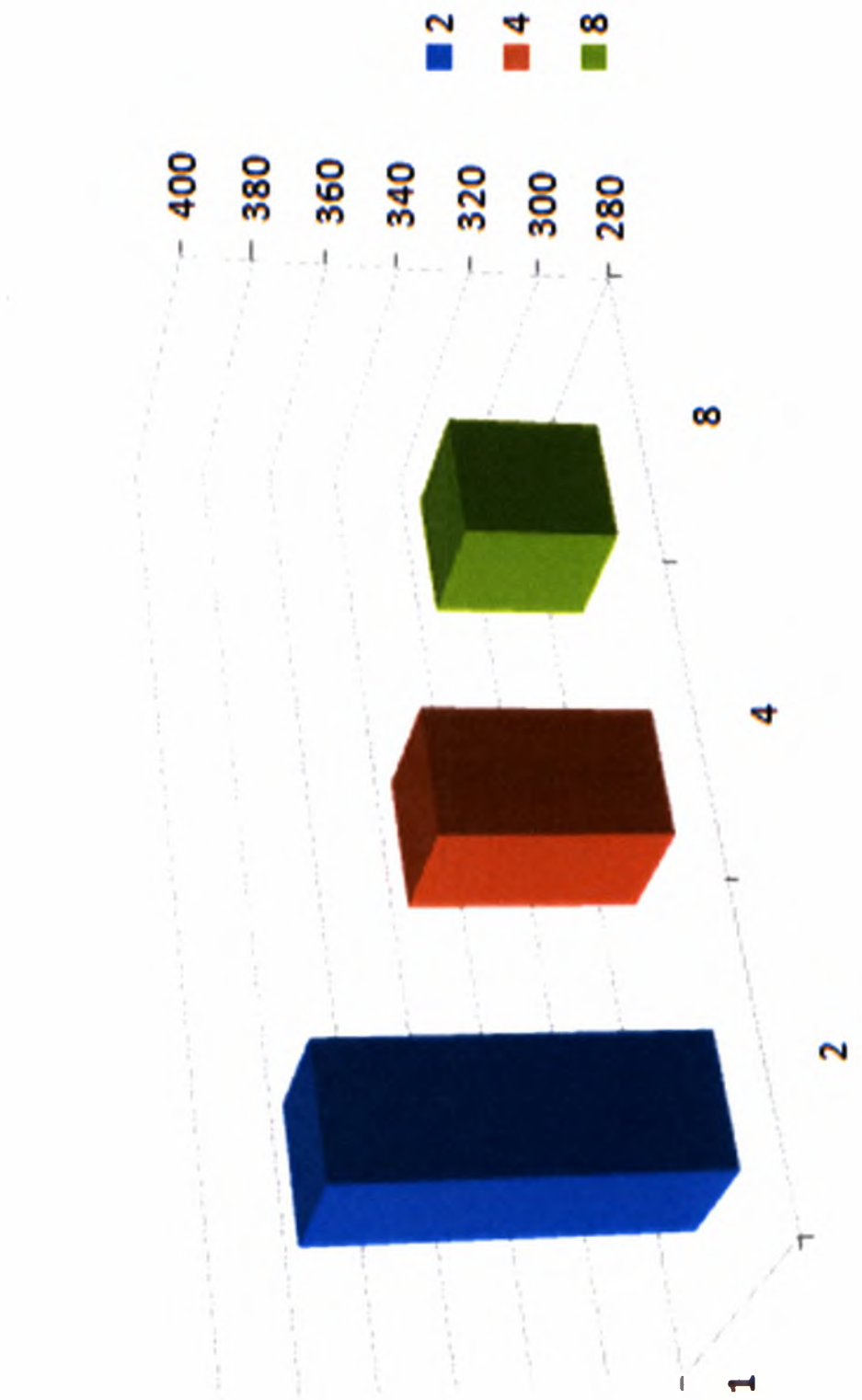
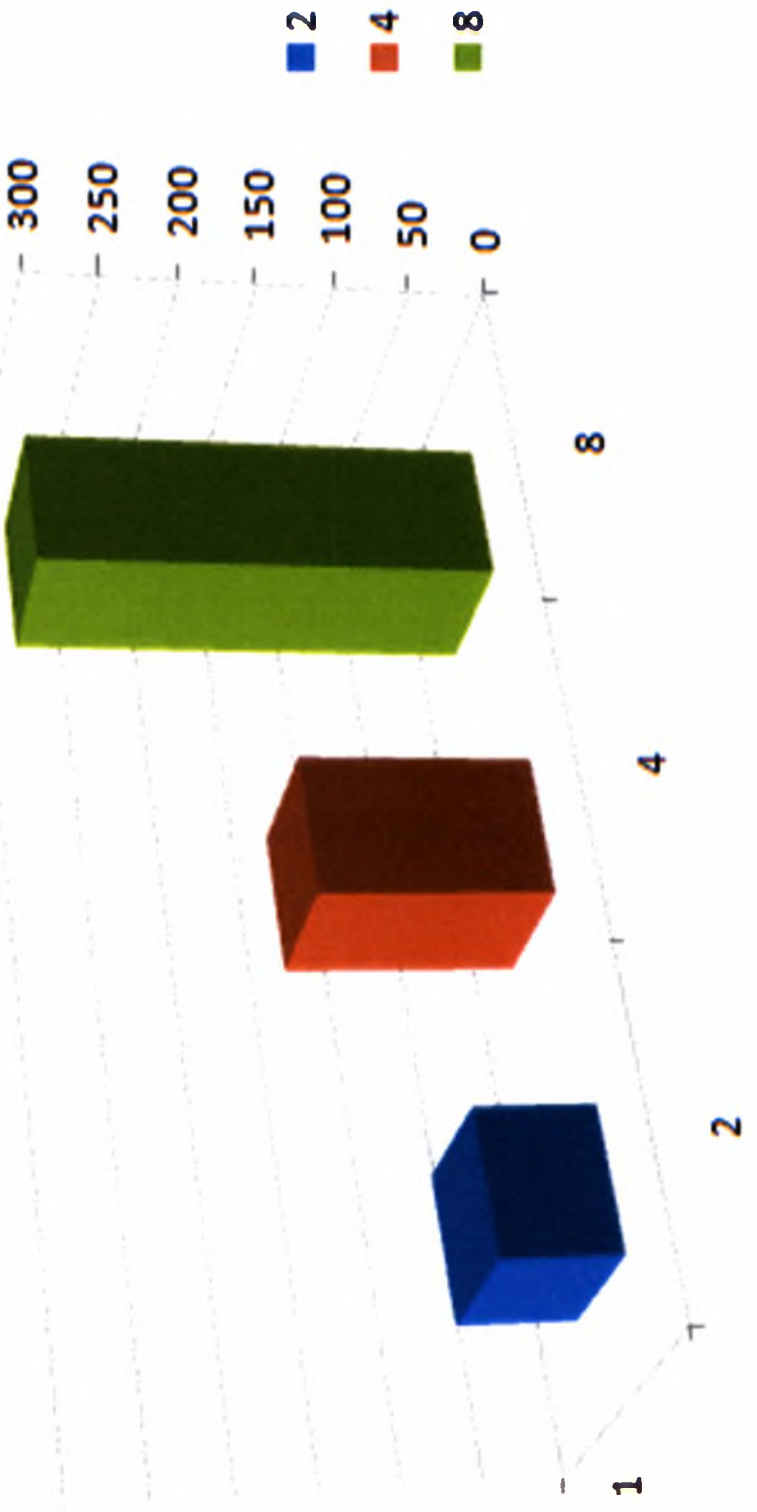


Figure 5.10: Binomial Method - Memory



Next we give the figures from the experimental results of simulations run on Centaurus, a cluster computer cited in Department of Computer and Communication Engineering of University of Thessaly. The options characteristics are the same as before. Centaurus consists of four nodes, each one of which consists of

- CPU: 4 x Intel Xeon 2.80 GHz
- RAM: 5GB
- HDD: 80GB ULTRA 360 SCSI
- LAN: 1 GBit

and runs Linux 2.6.16.13-4-smp with gcc version 4.1.0 (SUSE).

In Centaurus, the experimental results were in the same spirit as the previous ones. So, we decided to move one step with the Partial Differential Equations part, which in a first glance seem to go slower in a parallel mode. This is a result of the relatively slow inter-connectivity between the nodes and the LAM daemon. About the LAM daemon, it worth mentioning that its scheduling policy does not always take full advantage of the CPU, because it aims to a "fair" policy for many processes. Boosting LAM's performance was outside the context of this thesis. A better scheduling policy, though, combined by a better interconnection (since we have a large number of messages for Partial Differential Equations) would improve the results.

Because of all these reasons, we tried to "fool" LAM by increasing the system's overhead to its limits. So, we give the experimental results of the CPU time (and not the total time as before) needed to reach to the solution if 30 clients wanted to compute the same option price in the same way at the same time.

The results showed us an improvement of about 8% for the Explicit Finite-Differences scheme, 10% for the Implicit Finite-Differences scheme and 15% for the Crank Nicolson scheme when going from 2 to 4 processes. So, for a real time application, where more than 30 requests -or even more complicated ones- are expected, the program is expected to scale satisfactorily. For the other two methods, the results are more straight-forward since we have a clear boost of performance as shown in the respective figures.

For the Monte Carlo method, we have an improvement of 54.5% when going from 1 to 2 processes, 72.3% when going from 1 to 4 processes, 84.8% when going from 1 to 8 processes and 90.9% when going from 1 to 16 processes.

For the Binomial method, we have an improvement of 50.9% when going from 2 to 4 processes, 84.6% when going from 2 to 8 processes and 88.8% when going from 2 to 16 processes.

Figure 5.11: Monte Carlo - Time (Centaurus)

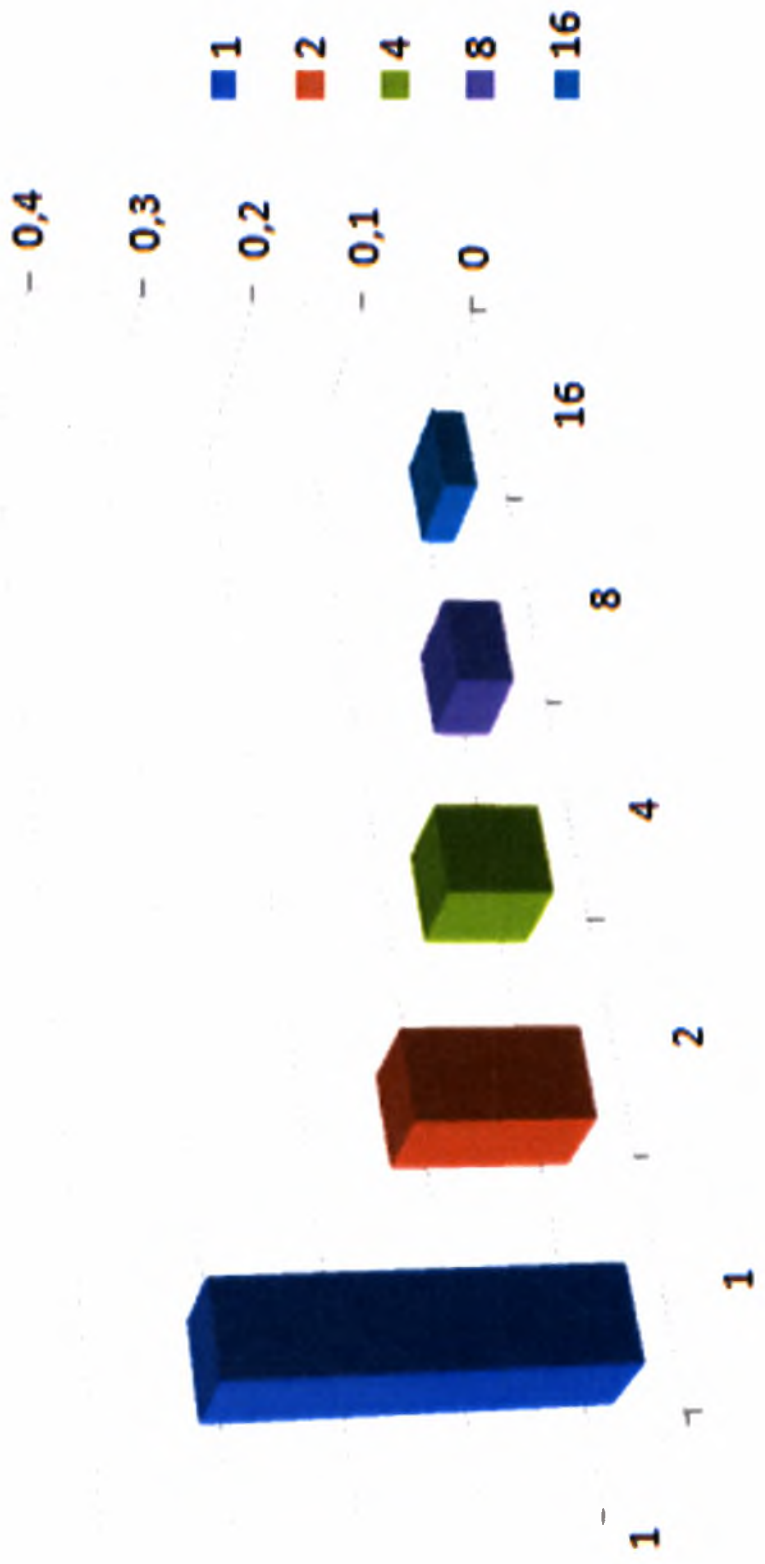


Figure 5.12: Explicit Finite-Differences - Time in Large Overhead (Centaurus)

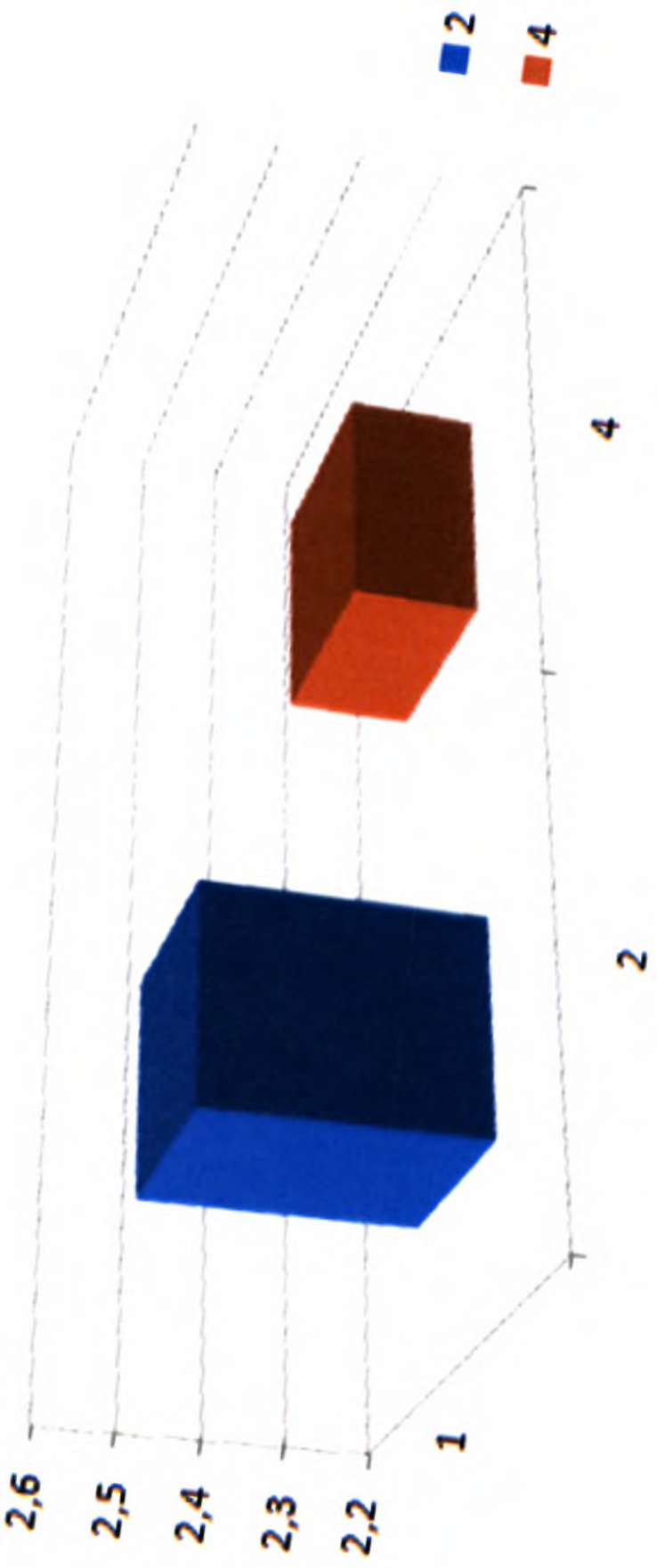


Figure 5.13: Implicit Finite-Differences - Time in Large Overhead (Centaurus)

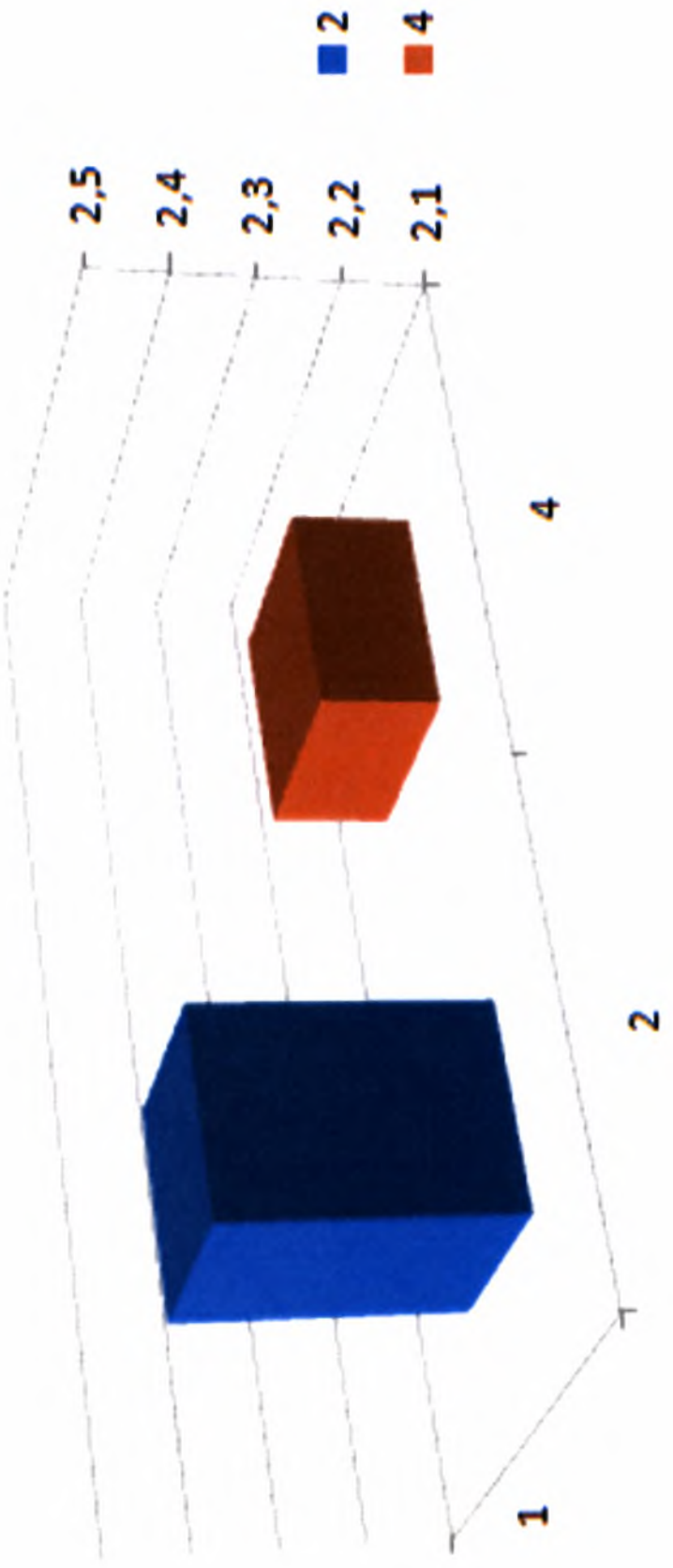


Figure 5.14: Crank-Nicolson - Time in Large Overhead (Centaurus)

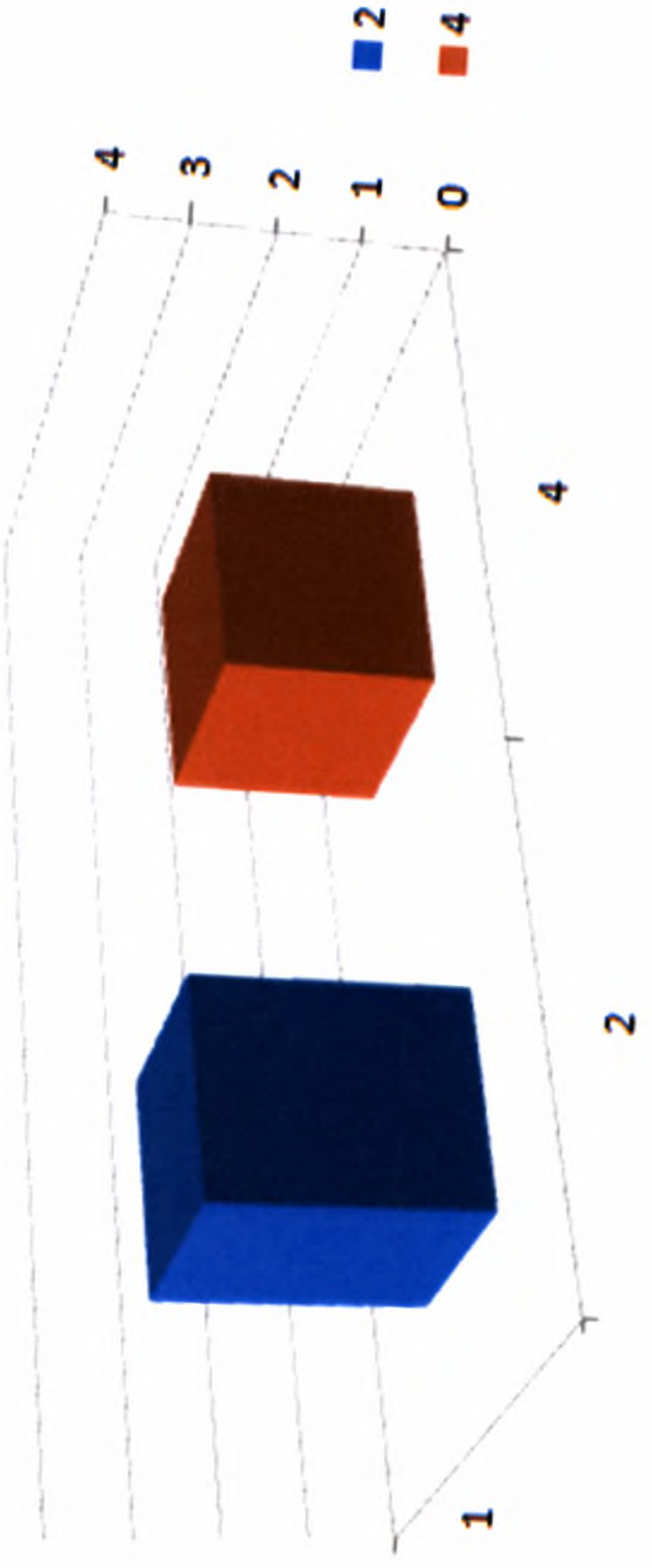
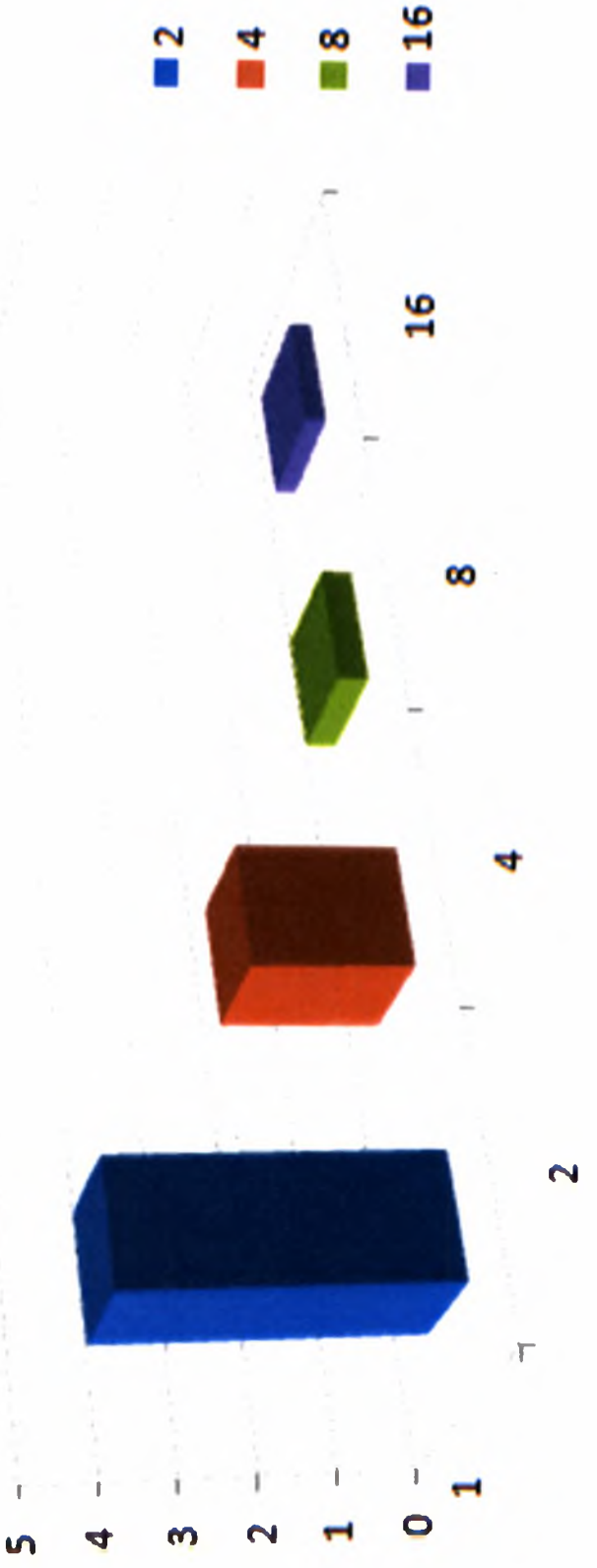


Figure 5.15: Binomial - Time (Centaurus)



Chapter 6

Conclusions - Future Work

The goal of this Thesis was to introduce the three most common methods for pricing financial derivatives and to extend them to run in a parallel way. Throughout Chapters 2, 3 and 4 we illustrated these methods and gave examples of their applications on European and American options. Especially for the European options, we gave algorithms for parallelizing these methods.

The study on these parallel algorithms, all along with the experiments ran on the two systems mentioned in Chapter 5, led us to the conclusion that the parallelization of these methods can give us results faster; either methods may have a better parallel performance than others but they can all take advantage of the benefits of parallel computations (faster execution, better CPU load balance, better time scheduling for many requests).

Of course, this is not all. Further work can and needs to be done. First of all, these algorithms could be tested in other parallel systems so that we could have a better picture of what happens and how the parallelization influences their efficiency under different environments.

Then, one could try to parallelize these methods for American options and check their performance. And then, move on to exotic and more complex options. As the computations complexity becomes bigger, the need for faster computations becomes even more essential and parallelism is our response to the demands of this problem.

Bibliography

[1] M. Bertocchi. Option evaluation techniques by parallel processing: A review, 1991. [cited at p. 12]

[2] F. Black and M.Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 1973. [cited at p. 28]

[3] Credit Suisse First Boston Corporation. Get real - using real options in security analysis. [cited at p. 94]

[4] Aswath Damodaran. The promise and peril of real options. [cited at p. 94]

[5] Alexandros V. Gerbessiotis. Architecture independent parallel binomial tree option price valuations, 2003. [cited at p. 66]

[6] Ilkay Boduroglu Halis Sak, Suleyman Ozekici. Parallel computing in asian option pricing, 2007. [cited at p. 22]

[7] http://www.wikipedia.org. [cited at p. 3, 93]

[8] Gary L. Mullen Jenny X. Li. Parallel computing of a quasi-monte carlo algorithm for valuing derivatives, 2000. [cited at p. 22]

[9] Didier Martineau Jerome Barraquand. Numerical valuation of high dimensional multivariate american securities, April 1994. [cited at p. 22, 24, 25]

[10] Suk Joon BYUN Jin Suk KIM. A parallel monte carlo simulation on cluster systems for financial derivatives pricing, 2005. [cited at p. 22]

[11] Mark Rubinstein John C. Cox, Stephen A. Ross. Option pricing: A simplified approach. *Journal of Financial Economics*, 1979. [cited at p. 55]

[12] Paul Glasserman Mark Broadie. Pricing american-style securities using simulation. *Journal of Economic Dynamics and Control, Elsevier*, 1997. [cited at p. 22, 25]

[13] Don L. McLeish. Monte carlo simulation and finance, September 2004. [cited at p. 19]

[14] Rebecca Carter Micheal B. Giles. Convergence analysis of crank-nicolson and rannacher time-marching, 2005. [cited at p. 51]

[15] Jeff Dewynne Paul Wilmott, Sam Howison. *The Mathematics of Financial Derivatives.* Cambridge University Press, 1995. [cited at p. 51, 53, 66]

[16] Giorgio Pauletto. Parallel monte carlo methods for derivative security pricing. [cited at p. 10]

[17] Paul Glasserman Phelim Boyle, Mark Broadie. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 1997. [cited at p. 12, 22]

[18] ClearSpeed Technology plc. Computational finance technical example, July 2007. [cited at p. 19]

[19] Eleftherios Syrrakos. *Xrimatistiriaka kai Epitokiaka Paragoga - Apotimisi kai Efarmoges.* Conceptum, 2000. [cited at p. 6]

[20] James A. Tilley. Valuing american options in a path simulation model. [cited at p. 22, 24]

[21] Jerome Spanier Yongzeng Lai. Applications of monte carlo/quasi-monte carlo methods in finance: Option pricing. [cited at p. 19]

[22] Tony Lelievre Yves Achdou, Olivier Bokanowsky. Partial differential equations in finance, 2007. [cited at p. 35]

Appendices

Appendix A

Risk Neutrality

Let's consider the Black-Scholes equation:

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

As we can see, the growth rate μ does not appear in A.1. Therefore, the value of an option only depends on the standard deviation of the asset price and not on its rate of growth. If we could construct a portfolio with a derivative product and the underlying asset in such a way that the random component can be eliminated, then the derivative product may be valued as if all the random walks involved were risk-neutral. This is why, we can replace the drift term in the stochastic differential equation for the asset return (in this case μ) with the interest rate r whenever it appears. This way, we can price the option by calculating the present value of its expected return at expiry with this modification to the random walk.

In order to value the option using these data, we work as follows. First of all, we know that the present value of any asset at time T is that value discounted by $e^{-r(T-t)}$. Then we do as we described above: We consider that the random walk for the return on S has a drift r and not μ. Moreover, since we know that S has a lognormal distribution, meaning that the probability density function of S is

$$\frac{1}{\sigma S\sqrt{2\pi t}} e^{-(f-f_0-(\mu-\frac{1}{2}\sigma^2)t)^2/2\sigma^2 t} \tag{A.2}$$

we can replace μ with r and then get a new probability density function. Finally, we calculate the expected value of the payoff $\Lambda(S)$ using this probability density function. We do so by multiplying $\Lambda(S)$ by the risk-neutral probability density function and integrate over all possible future values of the asset, from zero to

infinity. Then, we discount to get the present value of the option:

$$V(S,t) = \frac{e^{-r(T-t)}}{\sigma\sqrt{2\pi(T-t)}} \int_0^\infty e^{-\left(\log(S'/S)-(r-\frac{1}{2}\sigma^2)(T-t)\right)^2/2\sigma^2(T-t)} \Lambda(S') \frac{dS'}{S'} \quad \text{(A.3)}$$

The main drawback of this method is that it requires us to know the probability density function of the future asset values. This may be easy for a constant coefficient random, but in a more complicated model, we must first the distribution before integrating to calculate the expected return. Often, the calculation of a probability density function involves solving a partial differential equation equivalent to that satisfied by the option and the subsequent integration must be in general carried out numerically, as well. It is usually quicker to solve the option pricing equation directly. Moreover, when we come to American options, it is much more difficult to see how to implement the risk-neutral approach, while the direct approach via the partial differential equation for the option can be extended in a clear-cut way.

Appendix B

Real Options

A real option is the right, but not the obligation, to undertake some business decision, typically the option to make a capital investment[7]. Real options are particularly important for businesses with a few key characteristics. The first is smart and reputable management with access to capital. Managers must understand options, identify and create them, and appropriately exercise them. This contrasts with businesses which business leaders focused on maintaining the status quo or maximising near-term accounting earnings. Businesses that are market leaders are also attractive, as they often have the best information flow and richest opportunities -often linked to economies of scale and scope. Finally, real options are most applicable precisely where change is most evident.
The binomial model, illustrated in Chapter 4, is currently the most widely used method to value real options.

Although real options exist in many businesses, they are not always easy to identify. Real options can be classified into three main groups:

- Invest/Grow Options
- Defer/Learn Options
- Disinvest/Shrink Options

Invest/Grow Options

1. **Scale up.** These options are used by companies that expect their market to grow in the future, such as high technologies companies. In this case, these companies need to do some initial investments that are expected to bring profit in the future

2. **Switch up.** This option values the opportunity to switch products, process or plants given a shift in the underlying price or demand of inputs or outputs. In more detail, this means that a company can make a mixed deal about a product that is about to buy and use, so that there is the possibility to change in a range of similar products.

3. **Scope up.** This option values the opportunity to leverage an investment made in one industry into another, related industry. Practically this option can give the opportunity to a leading business in one field of a production chain to enter another field with an advantage.

Defer/Learn Options

1. **Study/Start.** This option gives its holder the opportunity to invest in a particular project in some time in the future. The holder can wait for some period before investing, which reduces uncertainty but, of course, costs some more. Practically, an investor would use such an option if he had seen that the field he wanted to invest in, would have grown adequately.

Disinvest/Shrink Options

1. **Scale Down.** This option gives a company the opportunity to shrink or downsize a project anytime while it is running. This means that if a project does not go well and does not bring the expected profit, it can be abandoned.

2. **Switch Down.** This option values the holder's ability to change to most profitable assets and investments as he receives more information.

3. **Scope Down.** This option values the opportunity of the holding company to abandon operations in a related field that do not bring the expected profit. This way, some money can be saved.

Real options can play an important role in the valuation of a company's stock price, especially for companies that compete in rapidly growing and highly uncertain markets. According to Credit Suisse [3], stocks of these companies are best viewed as a combination of the discounted cash flow value of the current, known business, plus a portfolio of real options. This real option can be estimated by taking the difference between the current equity value and the discounted cash flow value for the established businesses. Although there are analysts who disagree with that opinion, it seems to be rather accurate.

For a more detailed approach you could also refer to [4].

Appendix C

Implementation

Here, we will give some information about the implementation of this Thesis. First of all, the source code for the option evaluation application is written in C++. The application web interface is written in JSP and the network application used to establish communication between the JSP part and the C++ part is written in C.

In more details, there is a daemon (written in C) running on the host that also keeps the main computations program. This daemon listens for connections of two kinds: either for a new computations request or a request for results of completed computations. When a new request comes, then the daemon creates a separate process to serve the new client. The client enters the option characteristics in the proper JSP page and submits them. Then, the daemon receives this information, stores them in an xml file, associates a unique ID with the user and the data, sends the ID back to the user and calls the main computing program to execute the computations for the characteristics received by the client. The main program reads data from the xml file and writes output to another xml file. When the client wants to see the results, he opens the proper link and enters his code. Then, the daemon communicates again with the web interface and sends back any results available. Then, the JSP part prints these results on screen.

Next, we give some screen shots of the web interface of this financial toolbox. For any further information, you may refer to ac.anadiotis@gmail.com.

Figure C.1: Home Page

Figure C.2: European Options in General

Figure C.3: Binomial Model Form

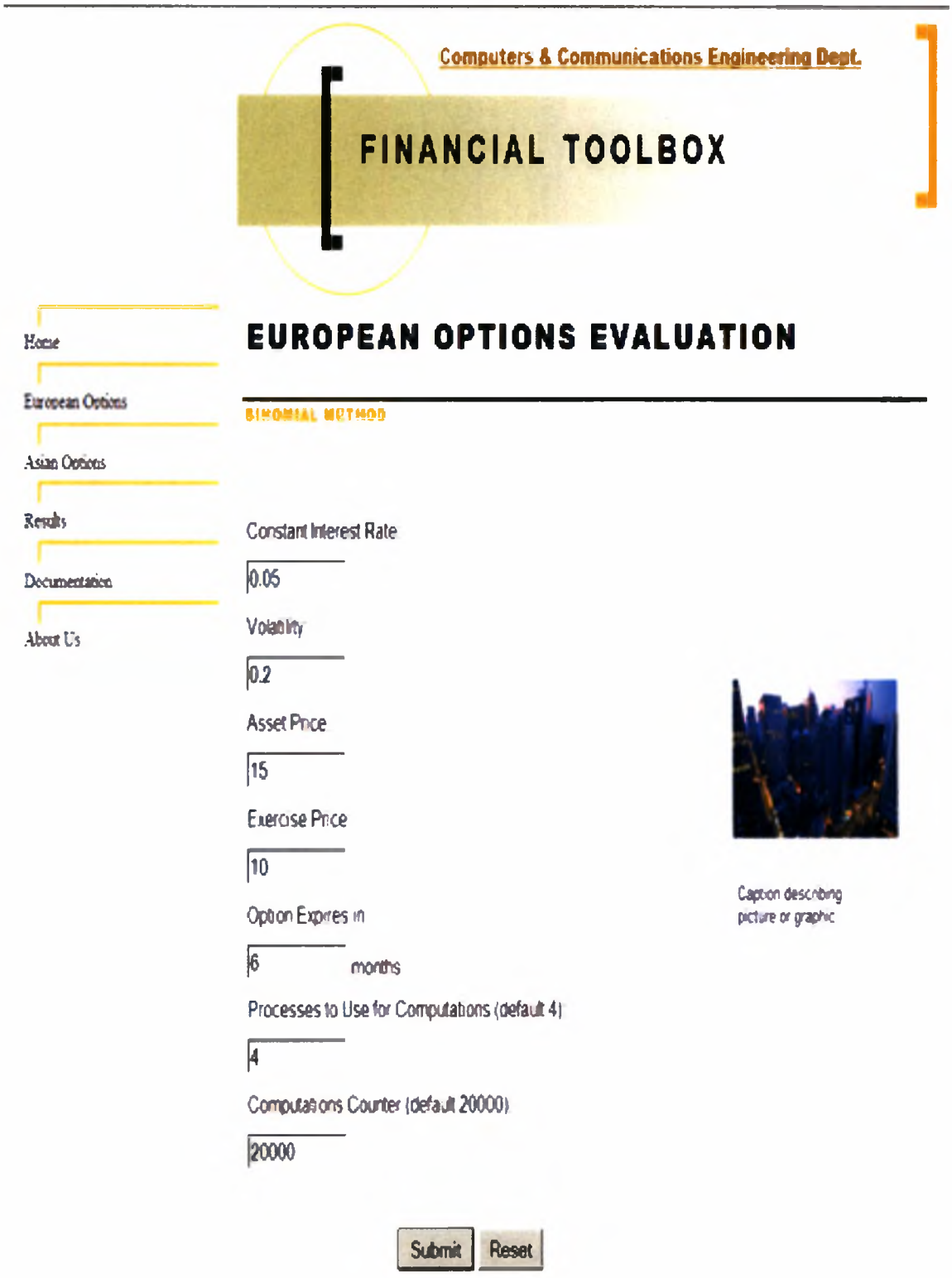


Figure C.4: Code Page

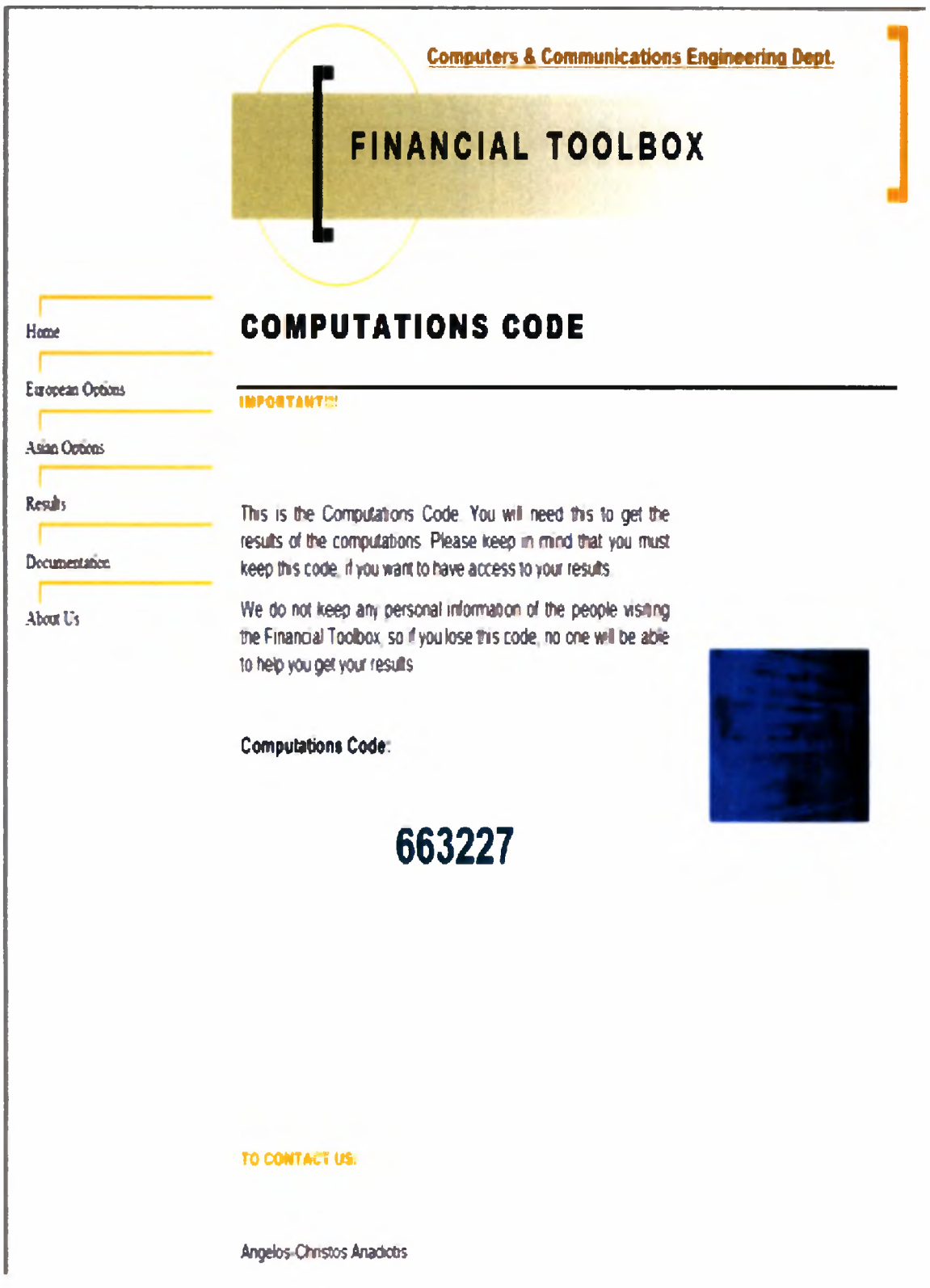


Figure C.5: Request for Results

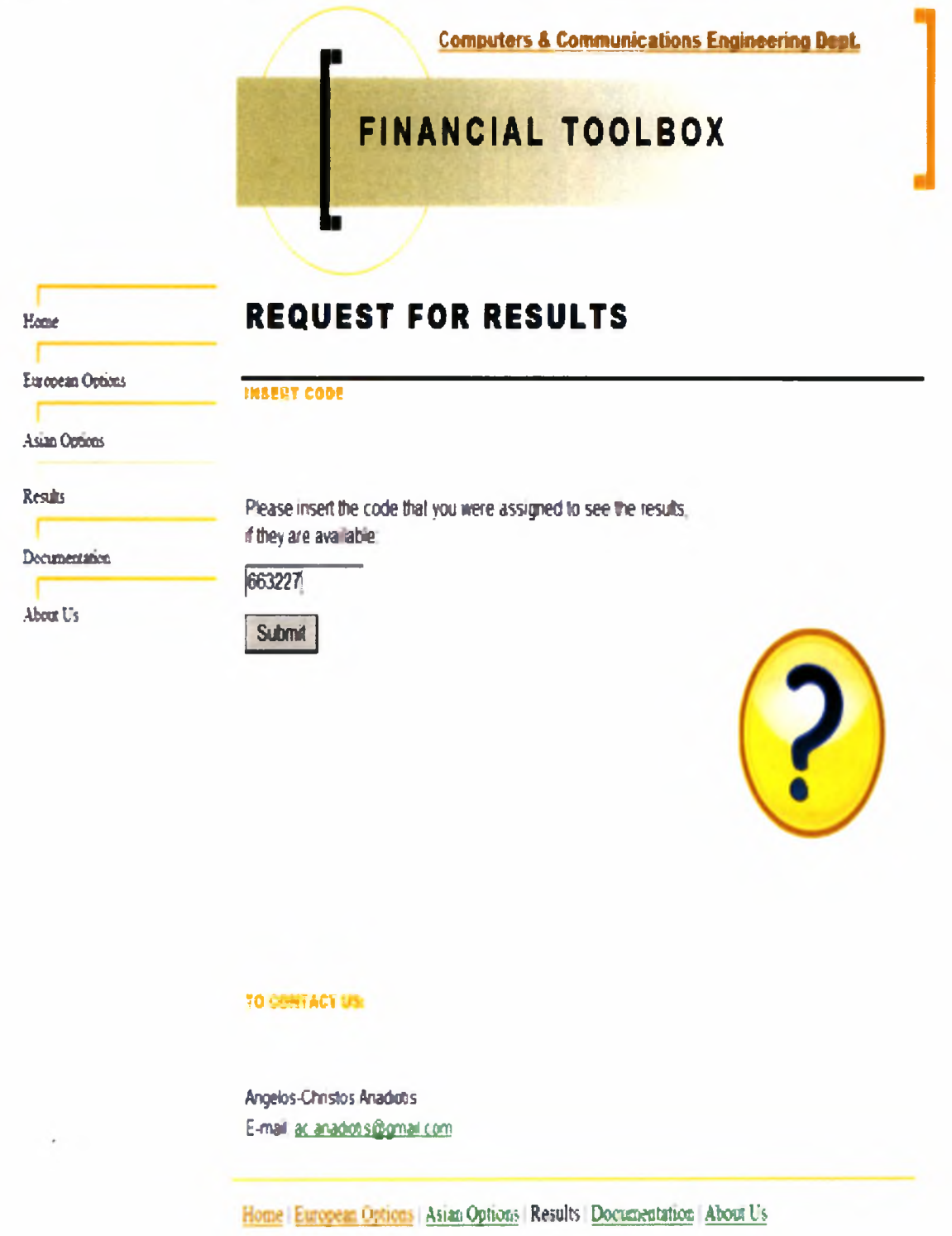


Figure C.6: Results for Partial Differential Equations

Computers & Communications Engineering Dept.

FINANCIAL TOOLBOX

Home

European Options

Asian Options

Documentation

About Us

RESULTS

PARTIAL DIFFERENTIAL EQUATIONS

ASSET PARTITION	OPTION PRICE
0	0.0000000000
1	0.0000000000
2	0.0000000001
3	0.0000000182
4	0.0000010642
5	0.0000331315
6	0.0005998464
7	0.0066571504
8	0.0467976404
9	0.2133718145
10	0.6487128602
11	1.3860417199
12	2.2896637019
13	3.2590174312
14	4.2499112346
15	5.2466229046

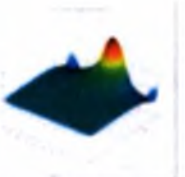


Figure C.7: About Page

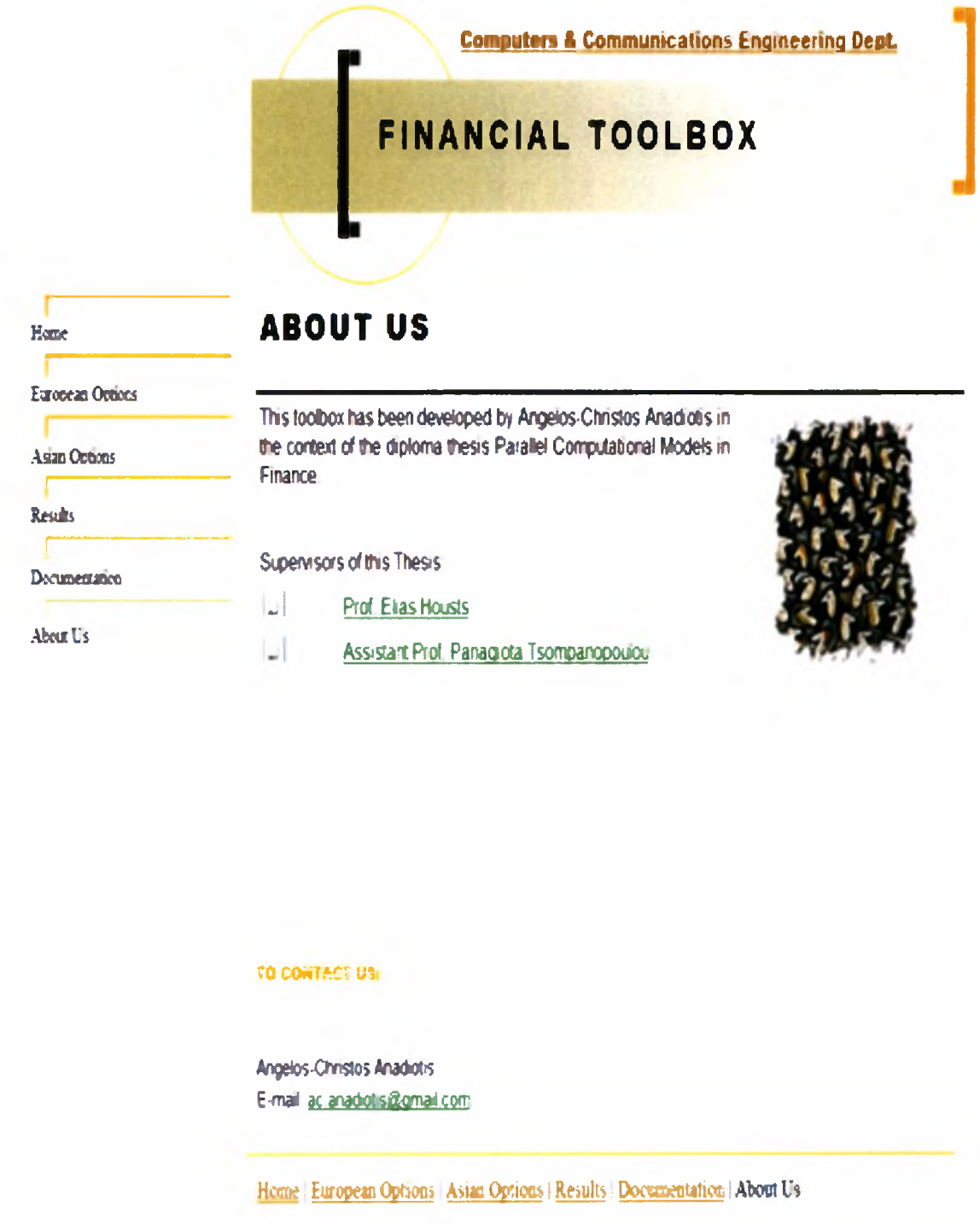
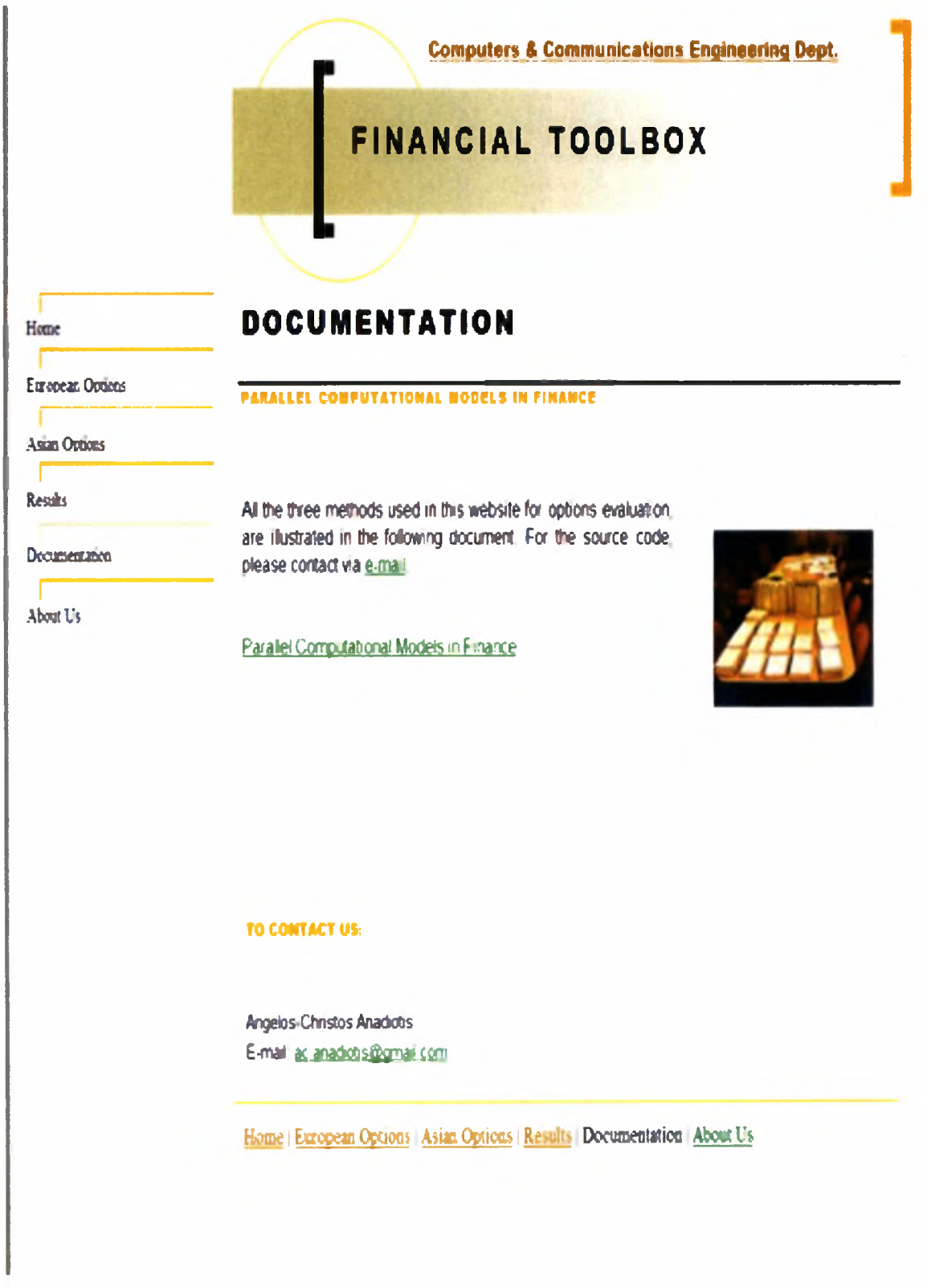


Figure C.8: Documentation



List of Symbols and Abbreviations

Abbreviation	Description	Definition
MPI	Message Passing Interface	page 61
LAM	Local Area Multicomputer	page 67