FLEET: A Framework for evaLuating European options in a parallEl and disTributed environment

by

Amit Chhabra

A thesis
submitted to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 2005

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

FLEET: A Framework for evaLuating European options in a parallEl and disTributed environment

BY

Amit Chhabra

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of

Manitoba in partial fulfillment of the requirement of the degree

Of

Master of Science

Abstract

Option pricing is one of the important problem in finance and demands efficient algorithms that produce accurate and fast results. This thesis aims to develop FLEET[1]: A Framework for evaLuating European options in a parallEl and disTributed environment on a Network of Computers (NoCs). In this thesis, the NoC is an interconnection of a collection of heterogeneous computers and an eight node shared memory machine. We have implemented a multithreaded pricing algorithm on shared memory architecture using Java OpenMP (JOMP [BWKO00, EPC]). FLEET uses the Common Object Request Broker Architecture (CORBA [SGR99, Bol02]) as a client-server model where a client requests for the value of an option (with certain characteristics of the option) over the network to a server. The server computes the option value using the Black-Scholes [BS73] model, a partial differential equation. The server is multithreaded and uses one thread-per-client policy to serve clients over the network. We use the explicit Forward-Time Central-Space (FTCS) finite-difference scheme to solve the Black-Scholes equation on the server side to evaluate the option price. We implemented a database containing the

[1]FLEET refers to a number of warships (processors in my case) working together under one command.

current stock information of the asset of interest, which can be accessed remotely. We compare and analyze the performance results using different scheduling technique on the shared memory machine with eight processors and achieve a speedup of approximately 4 running 16-threads. A key contribution is that the framework integrates the CORBA with the option pricing model.

This thesis is dedicated to my parents and my family.

Acknowledgement

Innumerable people have influenced this thesis.

First, my thesis committee members, Dr. Peter C. J. Graham and Dr. Ramanathan Sri Ranjan, whom I thank for their valuable comments and suggestions that helped to improve the quality of the final document.

I feel very privileged to have worked with my supervisors Dr. Ruppa K. Thulasiram (Tulsi) and Dr. Parimala Thulasiraman. I owe a great debt of gratitude to both of them for their guidance, inspiration, friendship and patience. Dr. Thulasiraman exposed me to the field of parallel computing in the early stage of my degree and I found it very interesting and challenging. Dr. Tulsi taught me a great deal about the field of high performance computing in computational finance. I want to thank both of them for the countless hours of discussion, advice and guidance regarding research, writing, speaking and life in general. Under their guidance, I learnt the subtleties of writing, the peculiarities of publishing papers in computer science and the importance of public speaking.

Their tireless editorial effort vastly improved the quality of this thesis and their valuable comments gave me confidence for my defence.

Most of all, this thesis would not have been possible without the support from my parents, my brothers, my sister-in-laws, and my nephew Rahat. I consider my second nephew, Varun, very lucky (atleast for me) as he was born on the day of my thesis defence. My father has been my inspiration all my life and he always stood by me and my decisions. The values, care and love given by my family are responsible for making me who I am today. I feel extremely lucky to have a family that has been very understanding and supportive of my studies. I also want to thank my uncle, aunty and their children for their care and support in Canada.

This thesis would not have been possible without the few good people who somehow crossed my life at the right moment in time, I would like to thank all my friends for their support, ideas and when I needed a timeout.

I would like to acknowledge the Department of Computer Science, the University of Manitoba for the Departmental Scholarship and partial financial support from the NSERC and University of Manitoba Research Grant Program (URGP).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Research in financial derivatives is one of the emerging areas of scientific computing. The price dynamics of the financial securities that the investor purchases determines the investor's overall budget. The investor selects an asset based on price, his budget and the market trend. Finance models used for evaluation and forecasting purposes to help the investor with the selection process typically lead to large dynamic, nonlinear problems that have to be solved in a short time span to beat the competition in the market place. The computational requirements for solving such models are huge and therefore, demand efficient algorithms and high performance computing capabilities [KNR00, Zen99].

Models of financial derivatives are generally manifested as a system of stochastic partial differential equations (PDEs). Solution of such systems on a sequential computer will require hours or may be even days depending on the size of the problem. Parallel computing speeds up the computational process by assigning tasks evenly to many processors. The processors co-operate and co-ordinate in the tasks to provide fast, reliable solutions.

An investor is not only concerned with the end result but also to obtain this result swiftly. The investors are not normally engineers or computer scientists. They could be ordinary people investing their money in a dynamic market environment. They are neither expected to know nor understand the underlying architecture of a computer or network nor should they need to understand the algorithm used to get their result. This leads easily to a client-server model, where the client allows the investor to request information from a server that transparently provides the required information. The clients could be distributed over Internet.

1.1 Basic Definitions

Options were introduced to protect investments from market risk and to increase the return from a portfolio. There are two parties in an option contract: the ***holder*** and the ***writer***. A holder has the right to purchase/sell the underlying asset at a set price, if he/she chooses, within a specified time. The option writer has the obligation to fulfill the holder's choice. Only the terms that are relevant to the current thesis work are described here, for all other types of options and their definition, please refer to [Hul02]. Buying and selling underlying assets through an option contract are known as Call and Put options.

- *Call Option:* A call option [Hul02] is a contract that gives the right to its holder (i.e. buyer) without creating an obligation, to *buy* a prespecified underlying asset at a predetermined price. Usually this right is created for a specific time period, e.g. six months. If the option can be exercised only at its expiration (i.e. the underlying asset can be purchased only at the end of the life of the option), the option is referred to as an European style Call Option (or *European Call*). If it can be exercised on any date before its maturity then the option is referred to as an American style Call Option (or *American Call*).

- *Put Option:* A put option [Hul02] is a contract that gives to its holder the right without creating the obligation, to *sell* a prespecified underlying asset at a predetermined price. If the option can be exercised only at its expiration date (i.e. the underlying asset can be sold only at the end of the life of the option), the option is referred to as an European style Put Option (or *European Put*). If it can be exercised on any date before its maturity then the option is referred to as an American style Put Option (or *American Put*).

- *Stock Price:* The stock price refers to the current price of a stock when the option is decided upon.

- *Strike Price:* The strike price (also referred to as the exercise price) is the agreed upon price of the underlying asset that will be used at the time of exercising the option.

- *Payoff:* If the option is exercised at sometime in the future, the payoff from the *call option* is the amount by which the stock price exceeds the strike price. Call options therefore become more valuable as the stock price increases. For a *put option*, the payoff at exercise is the amount by which the strike price exceeds the stock price. Thus, a put options become more valuable as the stock price decreases. In other words, put options behave in the opposite way when compared to call options.

- *Expiration Time:* The time of day at which an option expires on the expiration date. Both put and call options become more valuable if the time to expiration is long as the holder of an option with longer life has more exercise opportunities than the holder of an option with shorter life.

Let us consider an example to understand the concept of option pricing. Let us suppose that the stock price of XYZ stock is \$100 on May 1, 2005. We enter into a European call option contract to buy the stock at $108 after six months i.e. on November 1, 2005. On November 1, 2005, if the stock price is $115 then we can exercise the option i.e. buy the stock at $108 and by immediately selling it in the open market, we can make a profit of $7 per share. On the other hand if the stock price on November 1, 2005 is less than $108, we are not obligated to buy the stock.

1.2 FLEET

For this study, we have developed a software system to price options. It is a client-server based framework called FLEET. We call our framework FLEET, for two reasons. First, *"fleet"* refers to a number of warships (processors in our case) working together under one command. Second, *"fleet"* means to move swiftly and in this research we move quickly to a steady state solution using a parallel approach.

The focus of the research, therefore, is two fold: (I) developing a client-server framework for investors where (a) a client requests information over the network to a server; (b) the server provides the solution as fast as possible to the client and (II) developing a multithreaded algorithm for option pricing using Java OpenMP (JOMP) on a shared memory machine. The server providing the services is also a multithreaded server developed using Java threads. We assume the clients are in a distributed network where the computers are heterogeneous with different operating systems and computing speeds etc.. The underlying framework for FLEET is the Common Object Request Broker Architecture (CORBA) [SGR99, Bol02]. CORBA allows the clients to request information from the server by making a method invocation without knowing whether the targetted CORBA object is implemented in the same process or it is implemented in a separate process possibly on a remote machine. The information that the client requests from the server is the price of an option which the user wants to invest in. We have created a graphical user interface (GUI) that allows a user connected to the Internet to use the GUI to provide the required information (such as type of asset, expiration date of the option contract, strike price etc.) for pricing an option's value.

For option pricing, the typical Black-Scholes (B-S) [BS73] model (see [Hul02] for a thorough treatment of this model) is used. The B-S model is solved using the Forward-Time Central-Space (FTCS) finite-difference technique (discussed in detail in Chapter 5). Option pricing using the B-S model is a computationally intensive task that draws knowledge and ideas from computational fluid dynamics. Moreover, non-linearity in the B-S model and real-time solution requirements (to beat the competition in the market place) makes it further challenging to solve the B-S equation. For faster pricing of the options and to make real-time decisions, we chose to implement a multithreaded version of the FTCS method on a shared memory machine.

The clients communicate with the server requesting the value of an option. The server uses the capabilities of the symmetric multiprocessor (SMP) to compute the option value using JOMP [BWKO00, EPC]. JOMP is an API that provides directives, library routines and environment variables to convert sequential code written in Java into parallel code. Parallel computing is one way to solve a computationally intensive problem such as option pricing more quickly. Since SMP's have globally shared memory machine, the problem of data partitioning is alleviated. Partitioning is the process of dividing the data on multiple computers for concurrent execution. Through multithreading (multitasking within a single program), we exploit the concurrency of the algorithm and keep the system of processors busy, thereby improving efficiency. The current problem formulation follows the computational fluid dynamic principles in solving the B-S equation, we have solved the problem using the FTCS algorithm utilizing the multithreading features of JOMP[BWKO00]. The user chooses the stock of interest and then asks for the current stock price. The server then connects remotely to a database to display the value to the user. The selection of the database depends on the user's selection of the underlying asset.

1.3 Organization

The rest of this thesis is organized as follows. The background on parallel computing environments is provided in Chapter 2. The background and related work on computational finance which includes four different numerical techniques (binomial lattice, Monte-Carlo, fast Fourier transform and finite-difference techniques) is presented in Chapter 3. In Chapter 4, we discuss the role of CORBA in FLEET and how it is used in our research. We describe our multithreaded pricing algorithm that uses FTCS finite-difference method in Chapter 5. In Chapter 6, we show the effect of various parameters on the value of the option and present the experimental results. Finally, we present our conclusions and future work in Chapter 7.

Chapter 2

Background: Parallel Computing Environments

Parallel computing is the simultaneous execution of a task (split up and specially adapted) on multiple processors to obtain faster results. The term *parallel computer* (or parallel processor) is used for a computer with more than one processing element (PE) or Central Processing Unit (CPU). A system with thousands of such PEs is called a massively parallel computers. Parallel computing is also an umbrella term for a variety of architectures, including symmetric multiprocessing (SMP), clusters of symmetric multiprocessor systems and massively parallel processors (MPPs) [Bat80].

Types of parallel computers can be distinguished by the kind of interconnection network between the processors (or PEs); and between processors and memories. Flynn's taxonomy [GGKK03] coarsely classifies parallel computers according to whether all processors execute the same instructions over multiple data at the same time (**single instruction/multiple data** – SIMD) or each processor executes different instructions over multiple data (**multiple instruction/multiple data** – MIMD). In this chapter, we briefly describe Flynn's taxonomy and various parallel programming models.

2.1 Flynn's Taxonomy

Flynn classified the parallel (and serial) computers into four main categories according to the number of instruction streams (ISs) and data streams (DSs) per cycle: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD).

SISD refers to an architecture in which a single processor executes a single instruction stream (IS), to perform operations on the data stored in single memory. This architecture corresponds to the classical von Neumann machine or the sequential computer.

MISD is a type of parallel computing architecture where many functional units perform different operations on the same data. This model is very unconventional and the applications for this architecture are few.

As most of the applications designed today fall under SIMD or MIMD models, these models are described below.

2.1.1 Single Instruction Multiple Data

*S*ingle *I*nstruction *M*ultiple *D*ata (or SIMD) refers to a set of operations for efficiently handling large quantities of data in parallel, as in a vector processor or array processor. A SIMD machine consists of an array of PEs with their own local memories (LM) which are controlled by a global control unit as shown in Figure 2.1. Instructions from a program are broadcast to all PEs and each PE executes the same instruction in *synchrony* but operates on different data (in their own local memories). As a global clock is required to synchronize the operations in a SIMD machine, this is called a synchronous programming models.

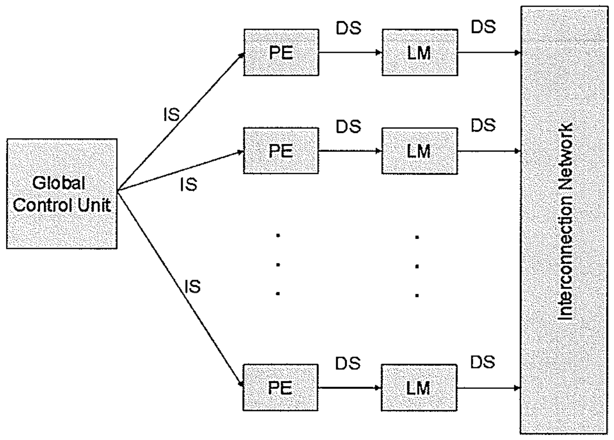

Figure 2.1: Logical view of SIMD model.

ILLIAC IV, see [BBK$^+$68], was the first SIMD machine developed at University of Illinois. SIMD machines became popular after the Cray Inc. [Inc] developed the first vector supercomputer in 1970s which used the SIMD instructions. MPP [Bat80] is another example of SIMD architecture.

2.1.2 Multiple Instruction Multiple Data

The ***M***ultiple ***I***nstruction ***M***ultiple ***D***ata (MIMD) machines, unlike SIMD machines, do not require the global control unit. Each PE in a MIMD machine has its own independent control unit (CU), thereby making it more flexible. All CPUs (PE and CU) perform different operations on different data and communicate through an interconnection network. Programming on these machines is more challenging as compared to shared memory machines as the programmer has to partition the data so that there is minimal or no communication. MIMD uses an asynchronous programming model. Examples of MIMD would be a multiprocessor or a network of workstations. A pictorial representation of the model is shown in Figure 2.2.

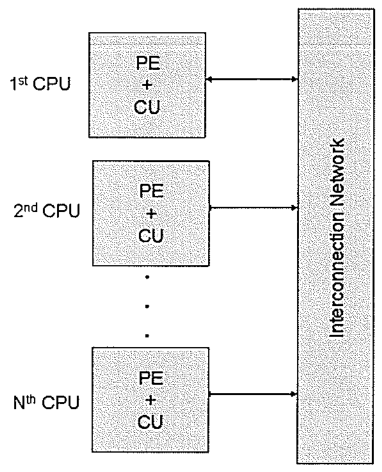

Figure 2.2: Logical view of MIMD model.

There is also a common extension to this model in which multiple autonomous processors simultaneously execute the same program (but at independent points) on different data. This is referred to as the "***S***ingle ***P***rogram, ***M***ultiple ***D***ata" or SPMD model. Depending on the memory organization, MIMD machines are classified into two categories namely **distributed memory multicomputers** and **shared memory multiple processors**.

2.2 Distributed Memory Multicomputers

A distributed memory multicomputer is an aggregation of several processing nodes that collaboratively execute a single computational task in a transparent and coherent way, so that they can be used, more or less, as a single, centralized system. The components of a distributed application are designed to operate separately but these components must be able to communicate. The CPU are connected through an interconnection network as shown in Figure 2.3 and since there is no global memory, the topology of the network determines the communication overhead.

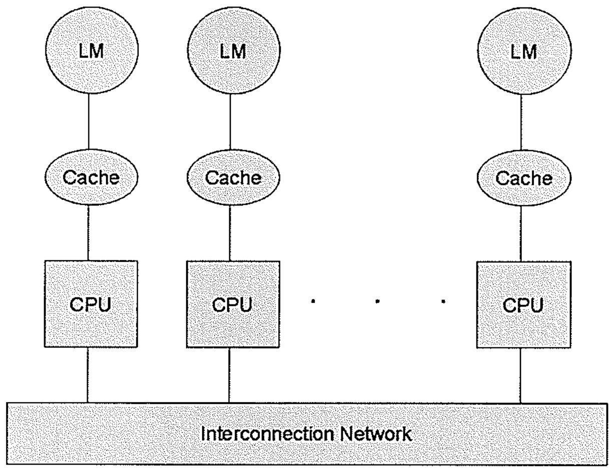

Figure 2.3: Logical view of a distributed memory machine.

The programmer is completely responsible for partitioning and distributing the data among the CPUs to improve the performance of the algorithm. Partitioning is the process of dividing the data on multiple computers for concurrent execution. As mentioned before, there is no global memory so, when necessary, the CPUs communicate by message passing. ***M***essage ***P***assing ***I***nterface (MPI) and ***P***arallel ***V***irtual ***M***achine (PVM) are two message passing libraries that can be used on a distributed memory machine. The Cosmic Cube [Sei85] and nCUBE [Cor87] are examples of such machines.

2.3 Shared Memory Multiprocessors

As the name implies, shared memory multiprocessors have access to a shared global memory [GGKK03] as shown in Figure 2.4. Unlike distributed memory multicomputers, in shared memory machines the programmer does not need to worry about partitioning, distributing and mapping the data. One of the drawbacks of the globally shared memory, however, is that locks and semaphores are necessary to properly synchronize the access to shared data. Examples of such machines include SGI Origin 3800 [Pub03] series.

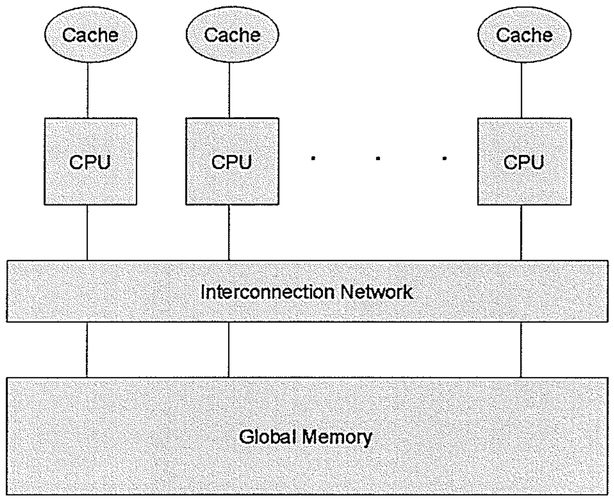

Figure 2.4: Logical view of a shared memory machine.

Based on the organization of the memory, shared memory multiprocessors can be classified as *Uniform Memory Access* (UMA); *Non-Uniform Memory Access* (NUMA); or *Cache-Only Memory Access* (COMA) models.

Each CPU in UMA model requires equal time to access any memory over the interconnection network (Figure 2.5). These machines are also commonly called *Symmetric Multiprocessors* (SMPs). Although it is easier to program on a machine using the UMA model, these machines do not scale well despite the fact that high speed caches are used to reduce access time to the memory.

In the NUMA model (Figure 2.6), the memory access time depends on the distance of memory from the CPU. Each CPU can access its own local memory faster than remote memory (the memory that is local to other PEs and is shared). NUMA machines are

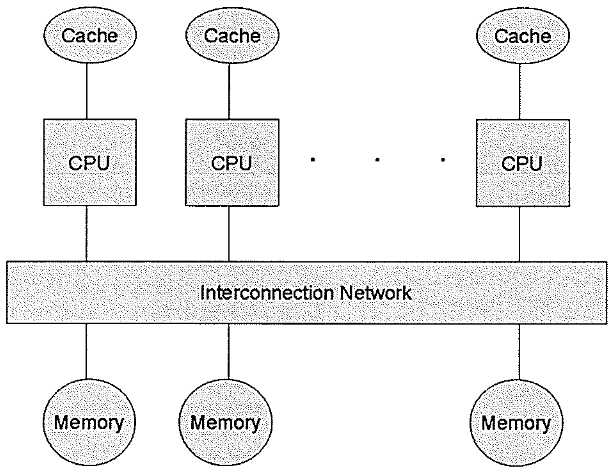

Figure 2.5: Uniform memory access model.

also sometimes called *distributed shared memory machines*. The remote memory access time increases with the number of CPUs, thus, the role of cache memory becomes very important. However, cache coherency problems may also arise. That is, two processors (processor 1 and processor 2) may have cached copies of the same variable x. When processor 1 updates the value of x and writes the new value to memory, processor 2 is unaware of this change. This causes processor 2 to have a stale value for x. To solve this cache coherency problem and to maintain stalability, techniques such as directory based protocols can be used. Such NUMA machines are usually called cache-coherent NUMA (CC-NUMA) machines. The SGI Origin 2800 series provides an example of a NUMA architecture.

COMA machines (Figure 2.7) are a variant of NUMA machines where only cache memories are prvided which are shared by all the PEs to form a global address space.

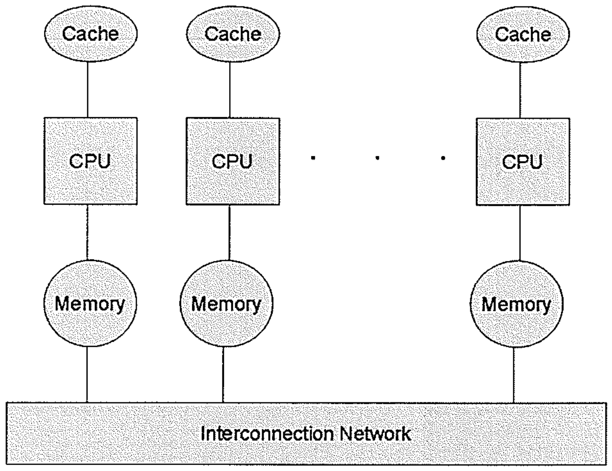

Figure 2.6: Non-uniform memory access model.

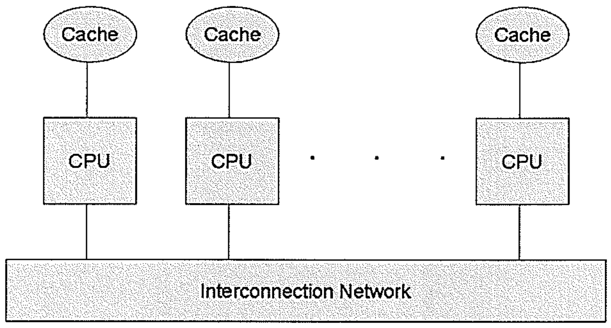

Figure 2.7: Cache-only memory access model.

2.4 Multithreading

In massively parallel processors [Bat80], there are two types of latencies which commonly hinder the performance of algorithms: *communication* and *synchronization* [AI87]. Communication latency arises when a processor requires data from another destination processor and must wait for it to arrive. Synchronization between processors is required until the data is ready at the destination processor. In both cases, processing resources are wasted by being inactive.

To overcome these latencies, either data dependencies have to be reduced or data locality has to be increased or both. The multithreading approach tries to hide the latencies by overlapping computations with communications. In multithreading, each task is divided into many threads where each thread is a sequence of instructions. The number of instructions per thread determine the granularity of the threads. The smaller the number of instructions per thread, the finer is granularity. Threads with large number of instructions are said to be coarse-grained.

There are many different types of multithreaded architectures, languages and libraries described in the literature. These differ in the granularity, memory model, etc. Some of them are explained below.

2.4.1 EARTH: A dataflow architecture

Dataflow computing [Den91] is a simple and powerful model for parallelism in a program. The first research prototype of a fine-grain dataflow computer was designed and built at University of Manchester [GKB87]. The attractiveness of the dataflow model of computation is that all forms of parallelism can be expressed using it. This makes dataflow a suitable environment for analysis of parallel algorithms, their sequential threads and resource requirement [GBT87, TB91]. EARTH (Efficient Architecture for Running THreads) [HMT$^+$95] is an architecture based on a multithreaded data-flow model of computation.

Two issues in general multiprocessor systems are the ability to tolerate synchronization/communication latencies and the capacity for scalable parallel performance in the presence of inter-processor communication. EARTH addresses these issues by, in part, implementing thread execution and synchronization with separate units. By using this approach, the incoming synchronization requests, which are kept in a special queue, need not interrupt the processing and this reduces the context-switching overhead since less context switches must take place. Also, long-latency operations, such as block moves, do not tie up the processor and this increases the performance of the system. It might seem that creating a separate synchronization unit would double the implementation cost. However, synchronization tasks are specific, do not require much processing power and can be easily implemented in a small amount of external hardware. In general, EARTH style multithreading supports tolerability, spawnability and mobility. The synchronization unit is designed to hide long and unpredictable latencies arising due to inter-processor communication. The threaded C language [HMT$^+$95] is used for programming on EARTH.

2.4.2 The Tera Computer System

Tera [ACC$^+$90] a multithraded, shared memory MIMD system, was the general-purpose parallel computer. It was specifically designed for use in tackling large scale applications such as reservoir simulation, seismic exploration, 3-D computer aided design, molecular modeling, etc. A Tera system [ACC$^+$90] can accommodate up to 256 processors and 128 hardware threads per processor.

The Tera processor architecture is an efficient fine-grained multithreaded architecture which performs a thread context switch every clock cycle. The major focus in the architecture is latency tolerance rather than latency reduction. The base of the Tera system architecture is the shared memory which allows the programmers and compilers to ensure that most data is physically close (available in the local memory) to the processor at the time of execution. Explicit-dependence lookahead technique facilitate the instruction level parallelism.

2.4.3 Cilk: An efficient multithreaded runtime system

Cilk (pronounced "silk") [BJK+95] is a C-based runtime system for multithreaded parallel programming. A multithreaded language provides programmers with a means to create and synchronize multiple computational threads, and the Cilk runtime system that automatically schedules the execution of these threads on the processors of a parallel computer. Cilk is a powerful language, it employs a work-stealing mechanism [BJK+95] that tries to allow all processors to be busy throughout the entire execution of the algorithm. Cilk is an algorithmic language, the parallel algorithm can be easily analyzed using two parameters, "work" and "critical path". Therefore, the user can efficiently judge the performance results with the theoretical results in Cilk.

2.4.4 Java

Java [OW99] provides threading for implementing coarse-grained multithreaded applications. Java threads can use all the features of the Java language. Although, Java supports multithreading on single and multiple processors, it is not a good choice for fine-grained parallelism as significant amount of time is required in context switching.

2.4.5 PThreads

PThreads is an abbreviation for POSIX threads and is implemented as a library for coarse-grained parallelism that provides POSIX-compliant functions for creating and manipulating threads. The library uses preemptive threads which increases the context switching time significantly on Linux and Unix systems.

2.4.6 OpenMP

The OpenMP organization [Org] refers to OpenMP as:

"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer."

Sequential code written in C/C++ and Fortran can easily be converted into multi-threaded code using directives, library routines and environment variables provided by OpenMP. The directives in OpenMP are treated at compile time making it easier for the programmer to write and compile the code. In a non-OpenMP environment, the compiler discards the directives as comments, so after compiling the code generated is sequential.

OpenMP uses control structures to exploit parallelism. OpenMP uses fork/join model for multiple threads and upon encountering a parallel construct, a block of code is executed concurrently by multiple threads. There are three types of work-sharing constructs (Figure 2.8) in OpenMP [R. 01] namely *for loops*; *section*; and *single*. In *for loop* work sharing, the threading can be fine-grained (allowing individual iterations of the loop to be executed by a thread) or coarse-grained (allowing a chunk of iterations of the loop to be executed by a thread). Each thread performs the same operations on different data sets following the SPMD model. This is also called "data parallelism". The *section* work sharing construct divides the work into separate, independent sections that are executed concurrently by a thread. This construct is suitable for "functional parallelism". The *single* work sharing construct serializes a section of code. It is useful for sections that are not thread safe.

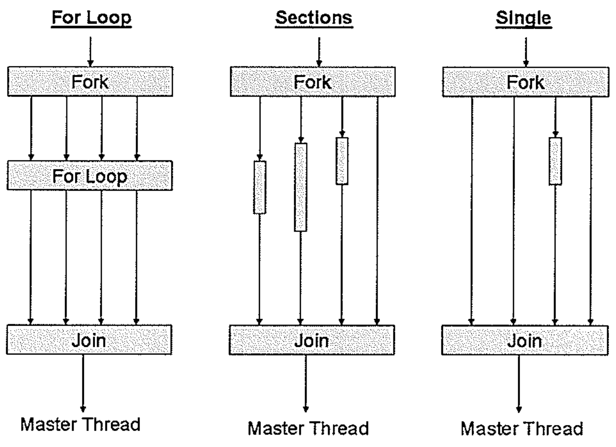

Figure 2.8: Work sharing constructs using fork/join in OpenMP.

An OpenMP program starts with a single thread (the master thread) and on encountering a parallel construct, it creates slave threads according to the requirements of the block (asequence of instructions). When a thread completes its work, it waits for the other related threads to finish. At the end of the block execution when all threads have finished their computations, the slave threads join together and the master thread continues its execution as a serial program.

Though the libraries and directives in OpenMP were designed for C/C++ and Fortran, EPCC [EPC] (a research group at the University of Edinburgh in UK) undertook a research project whose goal was to define and implement an OpenMP-like set of directives and library routines for shared memory parallel programming in Java. EPCC created a reference implementation, called JOMP (Java OpenMP) [Obd99, BWKO00, KOB01], that consists of a compiler and a runtime library. The compiler uses the in-line directives to translate Java source code into multithreaded code with calls to the runtime library. The whole system is pure Java and is thus platform independent. In this thesis, JOMP is used for the parallel implementation of the pricing algorithm and Java threads are thus used to implement the multithreaded server.

2.5 Summary

In this chapter, we provided a brief description of parallel computing environments. Starting with Flynn's taxonomy, we described distributed and shared memory architectures. We briefly introduced various multithreading paradigms including EARTH, Tera, Cilk, Java, PThreads and OpenMP. We will use these concepts in the following chapters where we design the parallel algorithm for option pricing.

Chapter 3

Background and Related Work on Computational Finance

Finance researchers have developed models that calculate the option premiums as a function of certain variables. In the early 1970's, Fischer Black, Myron Scholes [BS73], and Robert Merton made a major breakthrough in pricing of stock options by developing an model which is now known as Black-Scholes (B-S) model. The model makes simplifying assumptions about how the real world market works but it is, nevertheless, widely used by the market players and option traders. Changes in the financial market can take place very rapidly. Black Monday (October 19,1987, see [Hul02]) has shown that asset prices may change significantly in a very short time leading to financial calamities.

One of the important problems in finance is option pricing. Options on stocks were first traded on an organized exchange in 1973 at Chicago Board of Options Exchange (CBOE). Since then, there has been a dramatic increase in the options market. The underlying assets of options include stocks, stock indices, foreign currencies and commodities [Hul02]. Many algorithms using different numerical and analytical techniques have been developed for option pricing (see, for example [Bar04]) in the past. We next describe the concept of discounting in section 3.1. This is required for computing option price using any numerical or analytical technique. Some of the commonly used numerical techniques are discussed in the following subsections.

3.1 Discounting

Discounting is a term used in finance to identify the amount of money needed now for a future return. For example, in bonds, we pay the discounted amount of the face value of the bond set to expire at a future date. In simple terms, the amount needed to invest now to get a larger return at a future date is known as the discounted value. Mathematically, if M, R, and T are money (M) invested for a period of T at interest rate R, then

$$P = M + I$$

where I is interest accrued and P is the value of M at maturity. A small change in the money invested can be written as:

$$dM = MRdt$$
$$\frac{dM}{M} = Rdt$$
$$ln(M) = \int Rdt$$
$$ln(M) = Rt + c$$
$$M = ce^{Rt}$$

if $t = T, M = P$ (boundary condition)

hence, $P = ce^{RT}$ or $c = Pe^{-RT}$

Therefore, the money M required now for a future return of P is given by

$$M = Pe^{R(t-T)}$$

3.2 Numerical Techniques

Pricing techniques can be categorized according to whether they are analytical or numerical. A model is said to be closed form if it can be written as a parsimonious equation, and so one can obtain partial derivatives that are also analytic. Analytic solutions are generally provided for continuous-time models and are very rare. As the computational power of computers has increased in the recent past, numerical methods that were considered complicated about 10 years ago are now commonly being used to solve pricing problems on notebook computers. When an analytical solution cannot be obtained, which is true in most of the practical situations, a numerical technique must be used.

There are many numerical methods [Bar04, Hul02], including: (i) approximation techniques (such as Monte-Carlo Simulation, for example [RST04, Sri02]); (ii) lattice or tree based techniques (such as binomial, for example [TB05, TLN$^+$01]); (iii) fast Fourier transform (FFT) based techniques [Bar04, BTT04, CM99]; (iv) finite-difference techniques (for example [TR00, TZG04]). Any of these methods can be employed, depending on the user's requirement in terms of accuracy and on the availability of computational resources. Although, the binomial lattice and Monte-Carlo methods have been used predominantly, the finite-difference technique is gaining momentum among finance engineers.

In this chapter, we first describe the various numerical techniques used and review related work that has used the binomial lattice, Monte-Carlo and FFT techniques in subsections 3.2.1, 3.2.2 and 3.2.3, respectively, to illustrate the current trends in computational finance research. In subsection 3.2.4, we explain the finite-difference technique and try to summarize the related work done using the finite-difference technique for the option pricing problem.

3.2.1 The Binomial Lattice Techniques

The binomial lattice technique is a useful and intuitive techniques for pricing an option by constructing a binomial tree. In 1979, Cox, Ross and Rubinstein [CRR79] used this simple and intuitive technique for the first time to compute option price. Before going into the mathematical details of this technique, let us consider a very simple example [Hul02]: a stock is currently priced at \$100 and it is known that at the end of six months the stock price will either be \$104 or \$96. We are interested in evaluating the European call to buy the stock for \$102 in six months. If, after six months, the stock price is \$104, the value of the option will be \$2; on the other hand, if the stock price is \$96, the value of the option will be zero. This simple scenario is illustrated in Figure 3.1.

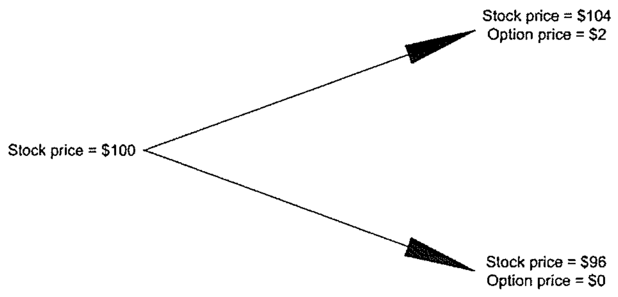

Figure 3.1: A simple example of stock price movement.

From the above example, it is clear that if no arbitrage[1] opportunities exist, we can set up a portfolio of a stock and an option in such a way that there is no uncertainty about the value of the portfolio at the end of six months (i.e. 0.5 years). We first build a portfolio with a long position (The state of actually owning a security, contract, or commodity) in Δ shares of stock and a short position (in future contract, the promise to sell a fixed amount of goods at a fixed price in the future) in one call option and then we can calculate

[1]The simultaneous purchase and selling of a security to profit from a differential in price. This might take place on different exchanges or in marketplaces.

the value of Δ which makes the portfolio riskless. If the stock price moves up from \$100 to \$104, the value of the shares is 104Δ and the value of the option is 2, so the total value of the portfolio becomes $104\Delta - 2$. If the stock price falls from \$100 to \$96, the value of the shares is 96Δ and the value of the option is zero, so the total value of the portfolio is 96Δ. For a portfolio to be *risk-neutral*, the value of Δ has to be chosen in such a way that the final value of the portfolio is the same for both alternatives. Therefore,

$$104\Delta - 2 = 96\Delta$$

or

$$\Delta = 0.25$$

This means, a riskless portfolio is:

Long: 0.25 shares

Short: 2 options

Thus, the value of the portfolio whether the stock price moves up or down is 24 (i.e. $104 * 0.25 - 2 = 24; 96 * 0.25 = 24$). Let the risk-free rate be 10% per annum; for the portfolio to be riskless, the value of the portfolio today must be the present value of 24 given by $24e^{-rT}$, where e^{-rT} is known as the discounting factor.

$$24e^{-0.10X0.50} = 22.829$$

If the stock price today is \$100 and suppose the option price is denoted by f, then the value of the portfolio today is $100 * 0.25 - f$ or $25 - f$. Therefore,

$$25 - f = 22.829$$

or

$$f = 2.171$$

This implies that, in absence of arbitrage opportunities, the current value of the option must be 2.171. It would cost less than 22.829, if the value of the option were more than 2.171 and would also earn more than the risk-free rate.

To generalize, consider a stock whose current price is S and the option on the stock whose current price is f. If T is the expiration date, the stock price can either move up from $S \rightarrow Su$ or drop down from $S \rightarrow Sd$ where $u > 1$ and $d < 1$. Similarly, if $S \rightarrow Su$, assume the payoff from the option is f_u; and if $S \rightarrow Sd$, assume the payoff from the option is f_d as illustrated in Figure 3.2.

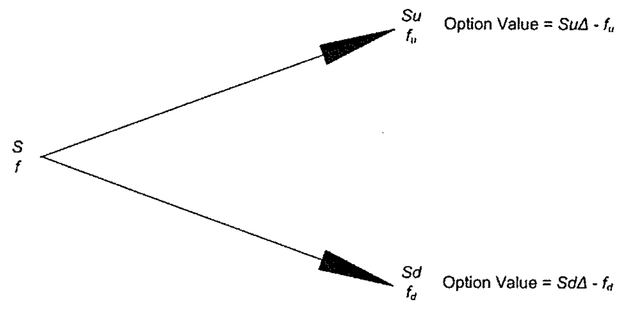

Figure 3.2: One-step tree of stock and option prices.

As before, let the portfolio consist of Δ shares of some stock in the long position and a short position in one call option, for the portfolio to be risk-neutral:

$$Su\Delta - f_u = Sd\Delta - f_d$$

or

$$\Delta = \frac{f_u - f_d}{S(u-d)} \tag{3.1}$$

Equation 3.1 gives the ratio between the change in the option price to the change in the stock price in time T. If r is the risk-free interest rate, the present value of the option becomes $(Su\Delta - f_u)e^{-rT}$ and the value of setting up the portfolio is $S\Delta - f$. Therefore,

$$f = e^{-rT}[pf_u + (1-p)f_d] \tag{3.2}$$

where

$$p = \frac{e^{rT} - d}{u - d}$$

Equation 3.2 calculates the option price of a one-step binomial lattice and the term $pf_u + (1-p)f_d$ calculates the expected future payoff.

We have, so far, only described binomial lattice technique with one step (i.e. the technique considers just one price movement during the entire period of the option contract, whereby the actual price behavior information is lost). To be more realistic, we must consider the stock price movements that are composed of large number of small price movement in the given contract period [CRR79] as shown in Figure 3.3. In this case the contract period T is divided into a large number of small time intervals of length $\Delta t = T/N$ where N is the number of steps.

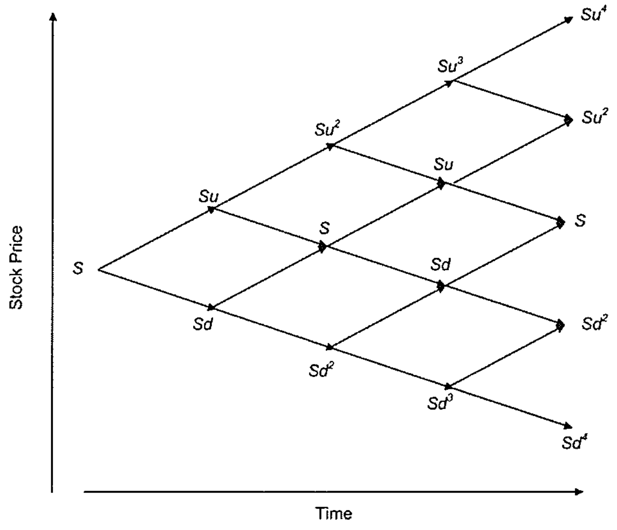

Figure 3.3: A multi-step binomial lattice.

Clark [Cla98] and Thulasiram et al. [TLN+01, TB02] developed parallel algorithm for the binomial lattice approach to preice options. Their techniques involve constructing a binomial lattice representing possible stock price movement from the present to the expiry date of the option in an intuitive fashion. Futher, Thulasiram and Bondarenko [TB02] have developed parallel lattice algorithms for options with multiple underlying assets and Huang and Thulasiram [HT05] have extended the study to price Asian options. Small machine size is a bottleneck when the problem size is large. On the other hand, multiprocessing does not always provide modest speedup for option pricing problems if inefficient algorithms are used (see, for example [Cla98]). Though lattice methods are perhaps the most widely used numerical method in finance, they are, in fact, a special case of simple explicit finite-difference schemes [Hul02, TR00].

3.2.2 Monte-Carlo Simulation Based Techniques

Monte-Carlo Simulation [Sok96] is a procedure in which random numbers are generated according to a probability distribution assumed to be associated with a source of uncertainty, such as a new products sales or, more appropriately for our purposes, stock prices, interest rates, exchange rates or commodity prices. Monte-Carlo simulation is a numerical approximation procedure that deliberately uses random numbers for the computation of option price. In this technique [Hul02], sampling is done to calculate the expected payoff in the risk-neutral world and then discounted at a risk-free interest rate. The accuracy of the results is determined by the number of simulations performed. Moreover, the simulations performed are independent of each other and hence can be implemented in parallel without any communication. This is known as an "embarrassingly parallel" problem.

The Monte-Carlo technique exploits the relationship between option prices and expectations to calculate the option prices from the simulation of asset prices. The basic underlying model is the B-S Model [BS73]. The steps to be followed to value the derivative with Monte-Carlo technique are described in [Hul02].

The Monte-Carlo simulation is advantageous when the payoff depends on the path followed by the underlying asset S or when it depends on the final value of S. One of the drawbacks, however, is that it is computationally very time consuming as many simulations have to be done to ensure accuracy. An implementation that uses truly-random numbers is called *crude Monte-Carlo method* whereas an implementation that uses pseudo-random numbers is called *quasi Monte-Carlo method.*

Srinivasan [Sri02] used the quasi Monte-Carlo simulation technique for option pricing while Rahmail et al. [RST05] used the crude Monte-Carlo simulation to study the effect of incorrect volatilities for underlying assets on option pricing errors. In Monte-Carlo simulations, increasing the sampling size increases the accuracy of the results. However, as the sampling size increases, computational cost also increases. Monte-Carlo methods are not generally regarded as well suited to the valuation of American options, since they do not capture the early exercise date precisely.

3.2.3 Fast Fourier Transform Based Techniques

Traditionally, the fast Fourier transform (FFT) has been applied to problems in science and engineering. Heston [Hes93] first used Fourier analysis for solving the option pricing problem analytically. Following this work, Scott [Sco97] and Bakshi and Chen [BC97] who used Fourier analysis to obtain the solution analytically and could not take advantage of the computational power of FFT. In addition, these analytical approaches were limited by their scalability due to the analytical integration needed to find the option value. It was Carr and Madan [CM99] who eventually provided a numerical technique to evaluate option price using FFT. In this section, we first describe how most authors have applied Fourier analysis to determine the option value followed by discussing Carr and Madan's [CM99] numerical FFT technique.

Consider a European call option for an underlying asset whose asset price at maturity (T) is S_T and whose characteristic function is given by:

$$\phi_T(u) = E[e^{ius_T}], \qquad s_T = lnS_T \tag{3.3}$$

where E is the expected return. Scott [Sco97] computed the risk-neutral probability of finishing in-the-money (a call option is in-the-money if the asset price of the option is greater than the strike price) using the above characteristic function as:

$$Pr(S_T > K) = \Pi_2 = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty Re[\frac{e^{-iuk\phi_T(u)}}{iu}]du \tag{3.4}$$

where K is the strike price and $k = lnK$. The delta of the option is numerically calculated as:

$$\Pi_1 = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty Re[\frac{e^{-iuk\phi_T(u-i)}}{iu\phi_T(-i)}]du \tag{3.5}$$

Assuming risk-neutrality, constant interest rate r and no dividends of the underlying asset, the option value is calculated as:

$$C = S\Pi_1 - Ke^{-rT}\Pi_2 \tag{3.6}$$

Due to the restriction of the integrand to its real part, the FFT cannot be applied to evaluate the integral [Bar04]. To overcome this drawback and to apply the FFT to the B-S model, Carr and Madan [CM99] introduced a numerical technique.

Using the fact that the call value is a function of strike price, Carr and Madan [CM99] define the call price function as:

$$C_T(k) = \frac{e^{-\alpha k}}{2\pi}\int_0^\infty e^{ivk}\psi_T(v)dv \tag{3.7}$$

where $\psi_T(v)$ is the Fourier transform of the call price, and k is the log of the strike price K. Therefore, the call option price needs to be calculated at different strike prices of the underlying asset in the contract. Discretizing equation 3.7 gives:

$$C_T(k) \approx M \sum_{j=0}^{N-1} \psi_T(v_j)\omega^{v_j k}\eta, k = 0, 1, \ldots, N-1 \tag{3.8}$$

where $M = e^{-\alpha k}/\pi$, $\omega = e^{-i}$ and v_j corresponds to various asset prices with η spacing. In this way, the FFT can be used for the option pricing problem.

Extending this model, Barua et al. [Bar04, BTT04] improved the mathematics behind the FFT of option pricing and have developed an efficient parallel algorithm to enable quicker pricing of options based on an earlier work, where the authors identified a one-to-one mapping between the Carr and Madan model and the popular Cooley-Tukey FFT algorithm [TT03]. Cerny [Cer04] explains the working of FFT in the familiar binomial option pricing model and how it relates to more complex continuous-time models routinely used in the finance industry.

3.2.4 Finite-Difference based Techniques

Finite-difference based techniques are gaining momentum among financial researchers for the option pricing problem. This technique involves the solving of PDEs manifesting the derivative under study. One such PDE which governs the evaluation of option price using finite-difference techniques is given by the B-S model [BS73]:

$$\frac{\partial C}{\partial t} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} = rC, \tag{3.9}$$

where C is the option price to be computed (the unknown variable), t is the time, σ is the volatility, S is the stock price of the underlying asset and r is the risk-free interest rate. This is an unsteady equation with $\frac{\partial C}{\partial t}$ being the unsteady term. The term $\frac{\partial C}{\partial S}$ is known as the *convective term* and $\frac{\partial^2 C}{\partial S^2}$ is known as the *diffusive term.* Diffusion is a process in a physical system that tries to mix the information in the system. In the current problem, a 2^{nd} order derivative term in the B-S model is a diffusive term which mixes the information arriving at the market place (captured by a first order derivative, the convective term). This knowledge helps to decide on a particular higher order accurate finite-difference schemes to be designed for the current work.

Finite-difference techniques involve the "discretization" of the above type of equation so that the dependent variables are considered to exist only at discrete points. The PDE is then transformed into a set of finite-difference equations (FDEs) that can be solved partitioning the solution space into rectangular meshes or 3D grids in certain cases [CTT04]. Hence, a problem involving differential calculus is transformed into an algebraic problem.

Solving the B-S model for option pricing numerically or analytically has been a challenging task for the researchers. However, the advent of high-speed digital computers and the ever-increasing computational power they offer has had a great impact on the application of finite-difference to various problems in engineering and sciences. The use of finite-difference methods (for example [Cla98] and [BS77]) to price options divides the solution space into rectangular meshes to represent time and the asset prices of the underlying asset in the options contract. The accuracy of the option value is determined by the size of the mesh in the temporal direction. Having finer meshes in the spatial direction (for asset prices) will allow pricing of the option for a multitude of market conditions. Thus, the fineness of the mesh contributes to the computational intensiveness of the problem and hence parallel algorithms are in great demand.

In the rest of this section, we briefly describe the basics of finite-differences and how the B-S model can be discretized to evaluate option price. The details of our proposed multithreaded algorithm for a specific finite-difference technique will be presented in

Chapter 5.

Assume for example, that we need to solve a PDE in which u (the unknown variable) is a function of x and y ($0 \leq x \leq 1, 0 \leq y \leq 1$), $u(x, y)$. Then we need to establish a grid on the domain by replacing $u(x, y)$ by $u(i\Delta x, j\Delta y)$. The points in the finite grid are located with the help of i and j and hence the difference equations are usually written in terms of (i, j). Assuming $u_{i,j}$ represents $u(x_0, y_0)$, the convention for labelling the points in the finite grid/mesh is illustrated in Figure 3.4. where the neighbours of $u_{i,j}$ are:

$$u_{i+1,j} = u(x_0 + \Delta x, y_0)$$

$$u_{i-1,j} = u(x_0 - \Delta x, y_0)$$

$$u_{i,j+1} = u(x_0, y_0 + \Delta y)$$

$$u_{i,j-1} = u(x_0, y_0 - \Delta y)$$

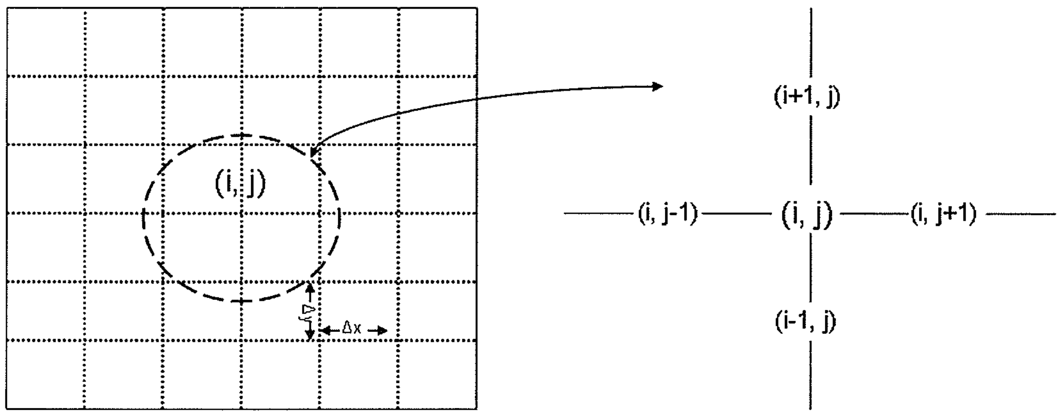

Figure 3.4: Labeling convention in a finite-difference grid.

To introduce the idea of finite-differences, recall the Taylor-series expansion (see for example [ATP84]) for the functions $u(x_0 + \Delta x, y_0)$ and $u(x_0 - \Delta x, y_0)$ at $x = x_0$ and $y = y_0$:

$$u(x_0 + \Delta x, y_0) = u(x_0, y_0) + \frac{\partial u}{\partial x}\Delta x + \frac{\partial^2 u}{\partial x^2}\frac{(\Delta x)^2}{2!} + \cdots \tag{3.10}$$

$$u(x_0 - \Delta x, y_0) = u(x_0, y_0) - \frac{\partial u}{\partial x}\Delta x + \frac{\partial^2 u}{\partial x^2}\frac{(\Delta x)^2}{2!} - \cdots \tag{3.11}$$

Adding equation 3.10 and equation 3.11, we get the central-difference form of the $2nd$ order term:

$$u(x_0 + \Delta x, y_0) + u(x_0 - \Delta x, y_0) = 2u(x_0, y_0) + \frac{\partial^2 u}{\partial x^2}(\Delta x)^2 + O(\Delta x)^4$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x_0 + \Delta x, y_0) - 2u(x_0, y_0) + u(x_0 - \Delta x, y_0)}{(\Delta x)^2}$$

The error in this case is of order 4. Switching the notation to i, j gives:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + error \tag{3.12}$$

Subtracting equation 3.11 from equation 3.10 and switching to notation i, j gives the central-difference form of the first order term with truncation error of the order of 3:

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + error \tag{3.13}$$

Similarly, the forward and backward approximations of the $1st$ order derivative can also be found by simply rearranging the terms for the Taylor series expansion of the function $u(x, y + \Delta y)$. Therefore, for an interior point (i, j) on the grid, $\partial u/\partial x$ can also be approximated by forward and backward approximation respectively as follows:

$$\frac{\partial u}{\partial x} = \frac{u_{i,j+1} - u_{i,j}}{\Delta x} \tag{3.14}$$

$$\frac{\partial u}{\partial x} = \frac{u_{i,j} - u_{i,j-1}}{\Delta x} \tag{3.15}$$

A large number of difference representations can be found for $1st$ order or $2nd$ order derivatives and then applied to the B-S PDE. For more details on finite-difference representations, readers are directed to [ATP84].

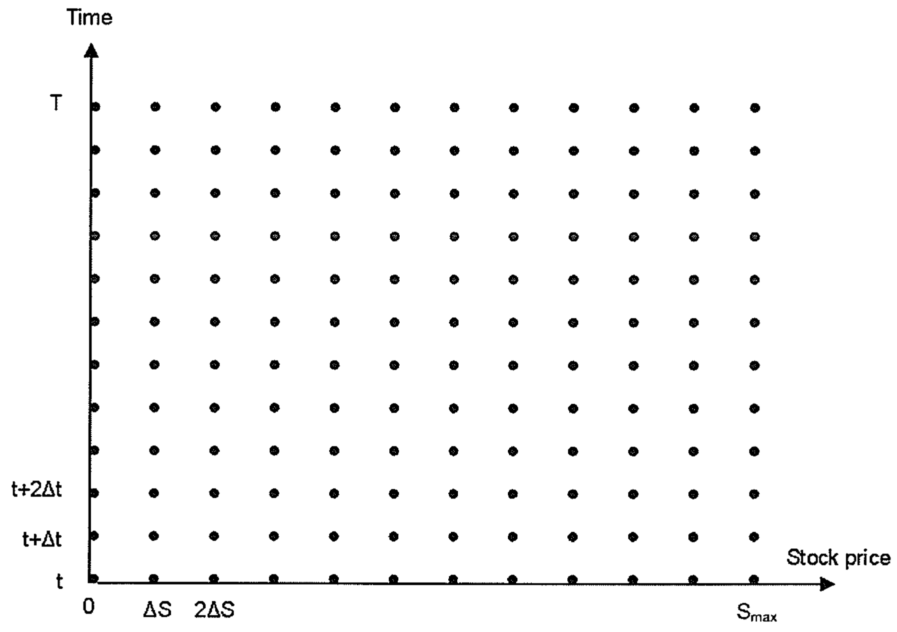

Figure 3.5: Time and Asset Price Grid.

The B-S model (equation 3.9) can be used to compute the option price using finite-difference method. If the expiration time of the option, T, is divided into N equally spaced time intervals of length $\Delta t = T/N$, a total of $N+1$ times points are considered:

$$0, \Delta t, 2\Delta t, \ldots, (N)\Delta t = T$$

Let S_{max} be sufficiently high stock price at the time of expiration. For example, is the stock price is \$100, then we take asset prices between \$0 and \$200 ΔS can be defined as $\Delta S = S_{max}/M$ and a total of $M+1$ equally spaced stock prices are considered:

$$0, \Delta S, 2\Delta S, \ldots, (M)\Delta S = S_{max}$$

Figure 3.5 depicts a finite-difference grid of asset prices and time containing a total of $(N+1)(M+1)$ grid points. The grid point (i,j) corresponds to time $i\Delta T$ and asset price $(j\Delta S)$. It is important to note that the pricing requires marching from the expiration date

$(T = (N+1)\Delta t)$ back to the current date $(T = 0)$. In other words, the index for time will decrease from $(N+1)\Delta t$ to 0 in steps of Δt.

There are two classifications of finite-difference methods: "explicit" and "implicit". Each of these methods have their own merits for accuracy, consistency and stability [Hof89]. For the *implicit scheme*, we consider the symmetric approximation for the term $\frac{\partial C}{\partial S}$ (in equation 3.9) which is obtained by taking the average of equations 3.14 and 3.15. For the term $\frac{\partial C}{\partial t}$ and $\frac{\partial^2 C}{\partial S^2}$ forward- and central-difference approximations are considered respectively:

$$\frac{\partial C}{\partial S} = \frac{C_{i,j+1} - C_{i,j-1}}{2\Delta S} \tag{3.16}$$

$$\frac{\partial C}{\partial t} = \frac{C_{i+1,j} - u_{i,j}}{\Delta t} \tag{3.17}$$

$$\frac{\partial^2 C}{\partial S^2} = \frac{C_{i,j+1} - 2C_{i,j} + C_{i,j-1}}{(\Delta S)^2} \tag{3.18}$$

Substituting the values of these derivatives into the unsteady equation 3.9 and rearranging the terms, the B-S equation can be rewritten as:

$$a_j C_{i,j-1} + b_j C_{i,j} + c_j C_{i,j+1} = C_{i+1,j} \tag{3.19}$$

where

$$a_j = \frac{1}{2} r j \Delta t - \frac{1}{2}\sigma^2 j^2 \Delta t$$
$$b_j = 1 + \sigma^2 j^2 \Delta t + r\Delta t$$
$$c_j = -\frac{1}{2} r j \Delta t - \frac{1}{2}\sigma^2 j^2 \Delta t$$

Thus, the B-S PDE has been replaced by an algebraic equation. A graphical representation of the grid points in equation 3.19 is shown in Figure 3.6. In the implicit scheme, there are three unknown variables (to compute the grid point $C_{i+1,j}$ one needs values of C at three other grid points which are unknown) that are computed by a set of coupled Finite Difference Equations (FDEs) for all grid points.

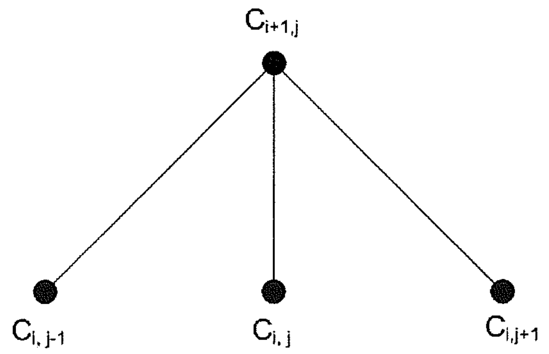

Figure 3.6: Grid points for implicit formulation.

For the *explicit scheme*, the approximations for the terms $\frac{\partial C}{\partial t}$ and $\frac{\partial^2 C}{\partial S^2}$ remain the same as the implicit scheme, however, the following approximation is used for the convective term $\frac{\partial C}{\partial S}$:

$$\frac{\partial C}{\partial S} = \frac{C_{i+1,j+1} - C_{i+1,j-1}}{2\Delta S} \tag{3.20}$$

The B-S equation for the explicit scheme can be written as:

$$C_{i,j} = a_j^* C_{i+1,j-1} + b_j^* C_{i+1,j} + c_j^* C_{i+1,j+1} \tag{3.21}$$

where

$$a_j^* = \frac{1}{1 + r\Delta t}\left(-\frac{1}{2}rj\Delta t + \frac{1}{2}\sigma^2 j^2 \Delta t\right)$$
$$b_j^* = \frac{1}{1 + r\Delta t}\left(1 - \sigma^2 j^2 \Delta t\right)$$
$$c_j^* = \frac{1}{1 + r\Delta t}\left(\frac{1}{2}rj\Delta t + \frac{1}{2}\sigma^2 j^2 \Delta t\right)$$

The formulation of the explicit FDE expresses one unknown in terms of three known values. A graphical representation of the grid points in equation 3.21 is shown in Figure 3.7. The best scheme for a given problem, however, is generally determined by the individual terms of the PDE manifesting the problem, and the required accuracy without violating the stability constraints.

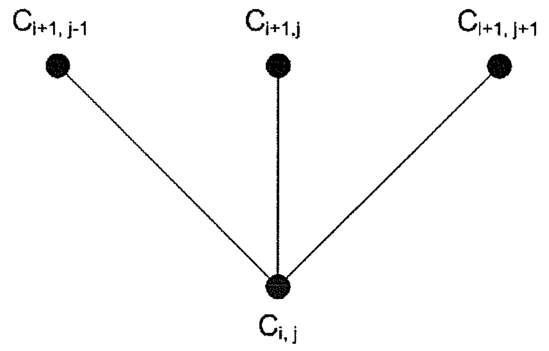

Figure 3.7: Grid points for explicit formulation.

Examples of the use of finite-difference methods for option pricing can be found in [Cla98] and [TZG04]. Although finite-difference methods are reasonably straightforward and the discretization of the problem domain is almost uniform, these methods are computationally intensive and produce less speedup as compared to the lattice methods [Cla98]. However, the accuracy of the option values from the finite-difference technique are typically much better than the lattice method. In this thesis, the finite-difference technique is used to evaluate option price.

Finite-difference and finite-element [ZFV98] techniques usually need more computations for complicated models. Finite-difference methods for the solution of parabolic PDEs have had broad application in several areas of science and engineering and provide accurate solutions of problems when appropriate boundary conditions are imposed. Applying boundary conditions sometimes becomes a difficult task.

Brennan and Schwartz [BS77] introduced a simple procedure using the standard implicit finite-difference scheme for the classic B-S PDE. Due to lack of computational power at that time, convergence of this method was, however, unclear. Jaillet, Lamberton and Lapeyre [JLL90] established the convergence of the Brennan and Schwartz method with a variational inequality formulation in a generalized B-S framework. The variational inequality approach not only ensured the convergence of option price but also the convergence of hedge factors [JLL90]. (The fact that the finance community does not look for help from other areas of research such as computational science is evident from the huge gap between the publication dates of these two works.)

Numerical considerations may require the use of a more sophisticated finite-difference method (for example, when the B-S model has little or no diffusion [ZFV98]). Coleman et al. [CLV02] introduced an algorithm that does not have any restrictions on the type of finite-difference method used; it is applicable to the discretized problem from the standard finite-difference to more sophisticated approximation schemes unlike the Brennan and Schwartz [BS77] method. Coleman et al. used the Newton process to examine the close relationship between the option values of the consecutive time steps.

Mayo [May00], for example, evaluated American options using the implicit finite-difference method. The algorithm gave fourth order accuracy in the log of the asset price and second order accuracy in time. Thulasiram et al. [TZG04] have designed a second order L_0 stable algorithm for the pricing problem. This algorithm achieves the same error bound as that of the traditional Crank-Nicholson scheme, (discussed in [ATP84]), while at the same time assures that the error will not propagate. Hence, their scheme is better than Crank-Nicholson scheme, an A_0 algorithm that guarantees only on error bounds. Coleman et al. [CLV02] have shown that the Crank-Nicolson scheme is typically unstable for partial differential problems.

Valuing options using lattice and finite-difference methods often demands additional computational effort to achieve higher accuracy of results with finer grids or lattices. Andricopoulos et al. [AWDN04] developed a method which curtails the price ranges of the underlying assets for which computations are carried out, achieving considerable savings of computational effort with virtually no loss in accuracy. Such shorter ranges of asset prices implies smaller volatility of the asset, which indicates to a well informed trader/investor the lesser value of the option. In other words, coarser grids on asset prices are good only for assets that are less volatile. Generally, as the expiration time approaches, the volatility of the asset price decreases, decreasing the value of the option. Therefore, having non-uniform grids in the spatial direction with coarser grids near the expiration time should reduce computational cost significantly. However, we do not consider non-uniform grids in this thesis. Finite-elements would be a natural choice for non-uniform grids or dynamically adaptive grids.

3.3 Summary

In this chapter, numerical techniques employed in option pricing and some of the related work were described. We will use the explicit finite-difference technique for our multi-threaded algorithm to price options, which we describe in Chapter 5. In the next chapter, we describe the use of CORBA in the current research.

Chapter 4

CORBA in FLEET

The Common Object Request Broker Architecture [SGR99, Bol02] (or CORBA) is an open standard from the Object Management Group (OMG) [Gro97] for communication between distributed objects over networks. It is a platform independent, programming language independent and vendor independent architecture and infrastructure CORBA-based programs communicate using the standard IIOP (Internet Inter-ORB Protocol) protocol for applications running on the Internet. The communication between the programs is done with the help of objects. A detailed description of CORBA objects, components and communication mechanism is provided in the following sections following some motivation for choosing CORBA.

4.1 Why CORBA?

Today's enterprises need flexible and open information systems. Most enterprises must cope with a wide range of technologies, operating systems, hardware platforms, and programming languages. Each of these is good at some important business task but all of them must work together for the business to function. CORBA provides foundation for flexible and open systems. It underlies some of the Internet's most successful e-business sites, and some of the worlds most complex and demanding enterprise information systems.

Large computer networks such as the Internet, the World Wide Web (WWW), and corporate intranets are generally heterogeneous in nature. These networks might be made up of mainframes, PC systems running various versions of Microsoft Windows, IBM OS/2, UNIX or Linux workstations and servers, and perhaps even devices such as telephone switches, robotic arms, or manufacturing test-beds. The diversity is not only in the networks but also the underlying network protocols connecting these systems: Ethernet, FDDI, ATM, TCP/IP, Novell Netware, and various remote procedure call (RPC) systems, for example [SGR99]. There are many factors at work that fuel the heterogeneity of large computer systems [Bol02, FN04]. Some of them include:

1. Legacy systems: A financial or other business establishment might decide to replace old technology with new. However, the company has to integrate the old and the new systems to work together and it is often very expensive to replace the old system entirely.

2. Changing technology: With the rapid growth of technology, companies may have to buy new technology. This may lead to heterogeneity.

3. Mergers: In the case of a merger, the companies involved have to undergo considerable change in terms of technology and integrating their respective computer networks and systems.

The Importance of CORBA

In a network, CORBA can be important for two main reasons:

1. It simplifies distributed computing, and

2. It is a development standard.

CORBA blends all differences between operating system, programming languages, and process locations using an object oriented paradigm. In addittion, as a standard it eliminates potential differences between software interfaces.

Some applications by nature are distributed. The application is distributed if a) data is distributed, b) computation is distributed and, or c) the users of the application are distributed. Fundamentally, the rapidly increasing extent of networks and applications are due to the need to share information and resources within and across diverse computing enterprises. Therefore, we need a solution to develop, deploy, and integrate systems in a distributed heterogeneous environment. CORBA is one such solution developed by the Object Management Group (OMG) [Gro97] that provides the necessary framework and API for developing distributed applications. In the current study, both the users and the data are distributed, therefore, CORBA is suitable for the development of FLEET.

There are many technologies that could have been used for the client-server model in FLEET. However, we present a tabular comparison of Java's Remote Method Invocation (RMI) with CORBA in table 4.1. Unlike CORBA, RMI allows Java developers to invoke object methods, and have them execute on remote Java Virtual Machines (JVMs).

Table 4.1: CORBA vs. RMI

CORBA	**RMI**
Services can be written in many different languages,executed on many different platforms, and accessed by any language with an interface definition language (IDL) mapping.	Can only operate with Java systems - no support for systems written in C++, Ada, Fortran, Cobol, and others (including future languages).
CORBA supports primitive data types, and a wider range of data structures, as parameters.	RMI provides limited support for data types and data structures.
CORBA is ideally suited to use with legacy systems, and to ensure that applications written now will be accessible in the future.	RMI does not provide any support for legacy systems, its tied only to platforms with Java support.
CORBA is an easy way to link objects and systems together	RMI only works for Java objects.

4.1.1 CORBA Objects

A "virtual" entity capable of being located by an Object Request Broker (ORB) and having client requests delivered to it is known as the CORBA object [Bol02, Gro97]. A CORBA object can be instantiated by one or more *Servants*[1]. Communication between CORBA-based programs occurs purely with the help of objects. Although, these objects can be implemented using any standard programming language, each object is clearly

[1]Servants are programming language entities that exists in the context of a server and contain code that implements a CORBA object.

defined by an Interface Definition Language (IDL) that describes the format of data sent and received. Every CORBA Object has an *object reference* [Bol02] which is an object's globally unique identity used by clients to invoke methods on the object.

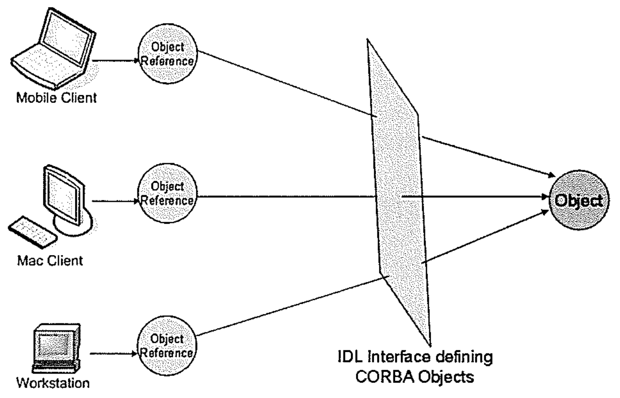

Figure 4.1: The virtual CORBA object.

Figure 4.1 shows that any part of the CORBA system can refer to a CORBA object using the object reference. However, the object is implemented in only one place on one of the servers in the system. The advantage of using an IDL interface to separate an object's use from its implementation is that any change in code will not affect the clients and also that objects can be made easily available across the network to a wide range of clients.

4.1.2 Object Request Brokers (ORBs)

An ORB [SGR99, Bol02] is a software component that mediates communication between clients and servers. The main function of an ORB is to deliver requests to server objects and return any responses to the client making the requests. The ORB hides the underlying complexity of the communication from the programmer by providing a proxy object in the client's address space, thereby creating an illusion that the remote object is local to the client machine. The ORB also hides the object location, implementation details (programming language, platform, and algorithm), the communication mechanism (IIOP, TCP/IP, local method invocation etc.), and whether the objects are active or not while making the request. We have used CORBA's naming service to implement location transparency.

Location Transparency

CORBA supports location transparency [Bol02] to make it easier to use. An object is location transparent if the client uses the same syntax to invoke an operation, no matter where the CORBA object resides (local or remote). For a client to find a CORBA object, it should know both the identity of the object and the location of the server process that provides a home for that object. A server must advertise its objects so that clients can find them. The server can either write the object reference to a file in a textual form or it can use the CORBA naming service [Bol02] to find an objects's reference dynamically given a persistent name for the objects. Using the naming service offers the following benefits:

- Clients can locate objects through standard names that are independent of the corresponding object's locations. This offers greater flexibility since one can modify the object's implementation or its location transparently from its users.
- The naming service provides a single repository for object references.

4.2 CORBA and its Components

In table 4.2, we present many of CORBA's components including Portable Object Adapter (POA) and General Inter-ORB Protocol (GIOP) and their functions. A pictorial view of

Table 4.2: Components of CORBA

Component	Function
ORB Core	to deliver requests to objects and return any responses to the clients making the requests.
IDL	OMG IDL is a declarative language for defining the interfaces of CORBA objects. All CORBA Objects must be described in OMG IDL.
Stubs	stub is a mechanism that creates and issues request on behalf of the client.
Skeletons	skeleton code translates the client's request and invokes the right method on the right implementation.
POA	keeps a reference to a specific object and serves as the glue between CORBA Object implementation and the ORB itself.
GIOP	defines the messages and format that are passed over the ORB between the client and the object.
IIOP	an implementation of the GIOP for TCP/IP.

a CORBA system is presented in Figure 4.2. An IDL is used to define the objects in CORBA and the IDL compiler generates client side stub and server side skeleton code.

4.3 The Role of CORBA in FLEET

CORBA will be used as the client-server model for the implementation of FLEET. Before computing the option price, clients will request the current stock price of a certain stock based on industry. The implemented CORBA object might reside in the same process

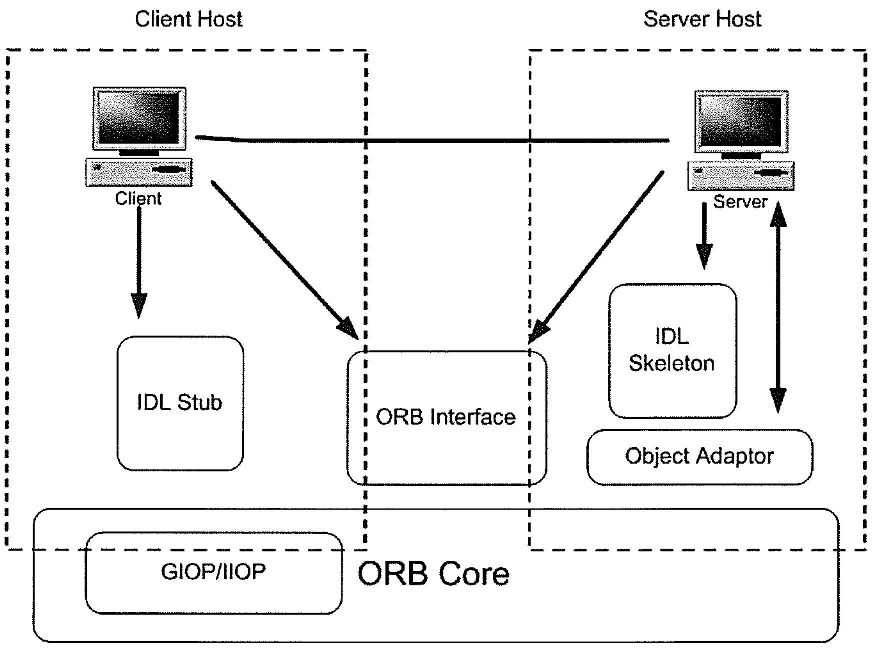

Figure 4.2: Components of a CORBA system.

or can be a part of a separate process on a remote machine. We use CORBA's naming service for location transparency. The naming service allows to associate names with the objects and hence a consistent style of coding is used to locate objects whether they are local or remote.

Figure 4.3 shows the general architecture used for evaluating option price and the shared memory multiprocessor [CTT04]. The Object Request Broker (ORB) core [Bol02] delivers requests for objects and returns any responses to the clients making the requests. To compute the option value, the data is sent to the compute server which is a shared memory multiprocessor machine running Java OpenMP (JOMP).

The common interface is defined using the IDL [Gro97]. We use Sun Microsystems' IDL-to-Java compiler. It generates the client-side stub and the server-side skeleton shown in Figure 4.3. When a client invokes an IDL-defined operation on an object reference

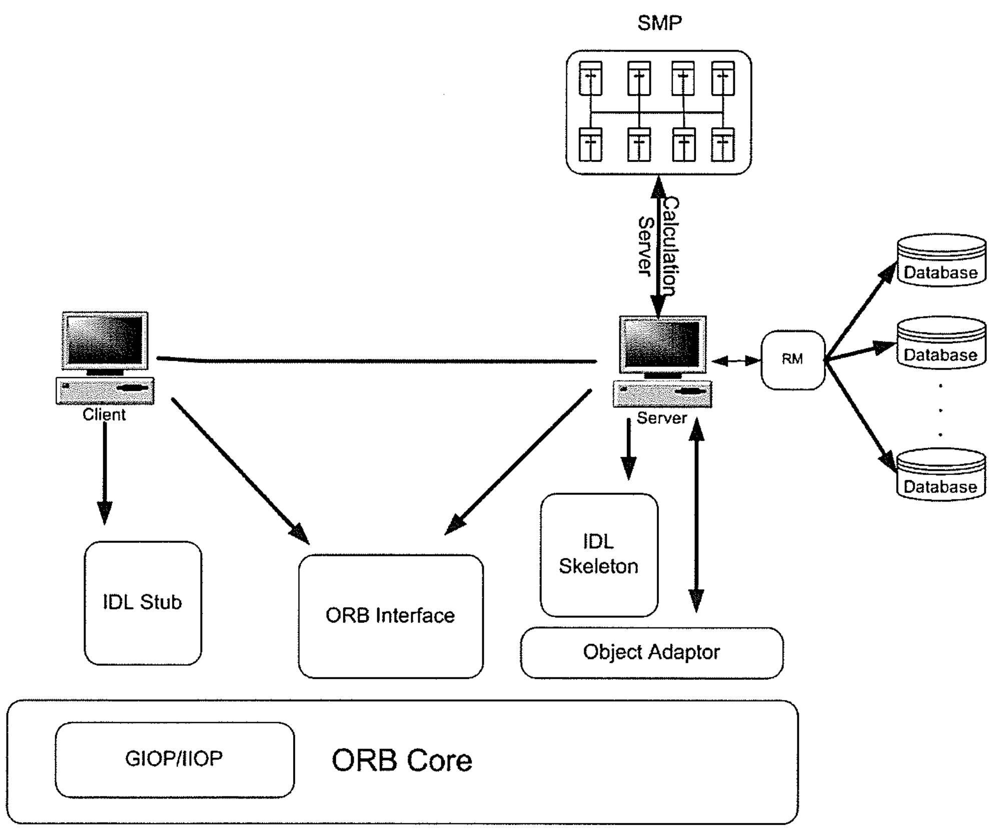

Figure 4.3: CORBA architecture for Option Pricing.

as if it were a local method, it must link in the stub code. The stub code creates and issues the remote request on behalf of the client. The stub handles the marshalling of a request, that is, converting data structure into wire format[2] [SGR99]. Once a request reaches the server side, the skeleton code is used to invoke the appropriate method of the object's implementation. Skeleton code translates wire format data into memory data format (unmarshaling) [SGR99]. The Object Adaptor serves as the glue between the CORBA object implementation and the ORB itself. It adapts the programming language concept of servants to the CORBA concept of objects and helps to dispatch incoming invocation requests to the correct servant. IIOP (Internet Inter-ORB Protocol) is used for the communication between the client and the server in different ORBs.

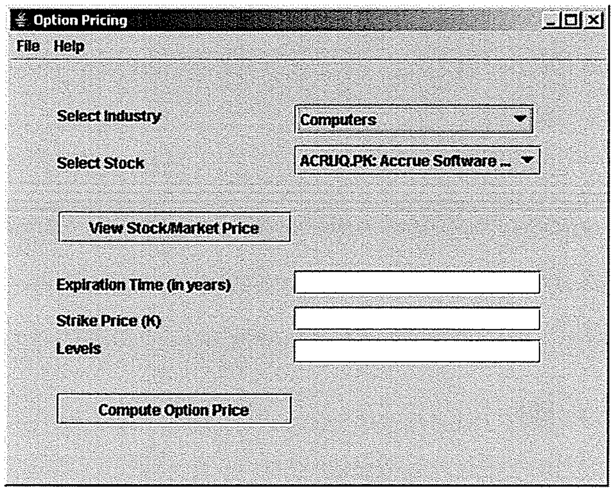

Figure 4.4: Client user interface.

Figure 4.4 shows the client's user interface. The user can select the different stocks based on industry. We have implemented a database of stock information based on industries which is remotely accessed by the SMP using the DB2 and SQL server running on

[2]The encoded format by which the request from the client is sent to the server-side through the ORB-core. The default wire format writes a byte which indicates the kind of object implementation being requested.

Neptunium and *Uranium* machines respectively in the Department of Computer Science. It should be noted here that the database of stock information is static. However, the database could be real-time if we got all the information from a reliable financial service provider. This model for having a multiple databases has been implemented keeping in mind that the information on all the stocks is not available at a single resource and hence we need to get the required data from multitude of distributed resources. Depending on the user's selection of the stock, the request is sent through the resource manager (RM) to the appropriate database and the stocks are populated in the list. Clicking on the "View Stock/Market Price", will establish a connection to the distributed database and the stock price of the asset will be displayed. Looking at the current market price, the user can decide whether he wants to compute the option value of that stock or wants to check the stock price of some other stock. The user is then asked to input the information required for option pricing such as the *expiration time* of the option contract, the *strike price*, and the *physical time steps* that will determine the desired accuracy of the computed option value. Larger numbers of levels leads to greater accuracy of the option value computed but at a higher computational cost. When the "*Compute Option Price*" button is pressed, it issues a request to the ORB core, which then looks for the servant that has the required object implementation and responds back to the client with the option value. The intermediate process is the computation of the option value on the SMP.

4.4 Summary

In this chapter, we described CORBA and how we used it for the implementation of option pricing system. Nowadays, some option price calculators are available online. Many financial and banking institutions offer some basic tools for evaluating options on the web. However, these sites neither mention the numerical technique used nor the volume of data processing involved. We have substantial amount of data processing and computation to ensure the accuracy of the option pricing results. This knowledge is essential to make informed decisions.

Chapter 5

Option Pricing using Finite - Differences

There are several finite-difference schemes available such as, forward-difference, backward-difference, central-difference, McCormack scheme, Crank-Nicolson, Richardson scheme etc. [ATP84]. The best scheme for a given problem is generally determined by the individual terms of the PDE manifesting the problemtogether with, the required accuracy and stability constraints. We have determined that for the reduced B-S equation, forward-differencing for the time (period of option) constraint and central-differencing for the space (prices) direction (FTCS) of the 3-D problem is much better to assure stability. Further, the accuracy of the results can be controlled by the use of a finer grid in the computational time direction as well as the space direction. That is, we iterate the solution process over many computational time steps until we reach a steady state solution.

The basic finite-difference concepts were introduced in section 3.2.4. In this Chapter, we will describe how the B-S equation can be reduced to apply the FTCS finite-difference method for the development of our multithreaded algorithm.

5.1 Reducing the Black-Scholes (B-S) Equation

In the derivation given below, C is the option price, S is the stock price, t is time, T is the expiration time, σ is volatility, r is the interest rate, E is the strike price, τ is the step size in the time direction (temporal) and x is the step size in the space direction (spatial).

Consider the B-S model,

$$\frac{\partial C}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} - rC = 0 \tag{5.1}$$

with initial and boundary conditions, $C(0,t) = 0$; and $\lim_{S\to\infty} C(S,t) = S$, $C(S,T) = max(S-E,0)$ for a call option and $C(S,T) = max(E-S,0)$ for a put option. For brevity, we will only consider a call option here. To transform the B-S model into a diffusion equation [WHD95], we set $S = Ee^x$, $t = T - \tau/\frac{1}{2}\sigma^2$ and $C = Ev(x,\tau)$. This results [Hul02] in equation

$$\frac{\partial v}{\partial \tau} = \frac{\partial^2 v}{\partial x^2} + (k-1)\frac{\partial v}{\partial x} - kv \tag{5.2}$$

where $k = r/\frac{1}{2}\sigma^2$. The initial condition now becomes

$$v(x,0) = max(e^x - 1, 0)$$

Equation 5.2 now looks almost like a diffusion equation and we can turn it completely into a diffusion equation, if we put

$$v = e^{\alpha x + \beta\tau}u(x,\tau),$$

for some constants α and β to be found. Then differentiation of this equation gives

$$\beta u + \frac{\partial u}{\partial \tau} = \alpha^2 u + 2\alpha\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} + (k-1)(\alpha u + \frac{\partial u}{\partial x}) - ku.$$

We can eliminate all the terms containing u by choosing,

$$\beta = \alpha^2 + (k-1)\alpha - k,$$

and by setting

$$2\alpha + (k-1) = 0$$

we can eliminate the $\partial u/\partial x$ term as well. Substituting the values of α and β, v can be rewritten as $v = e^{-\frac{1}{2}(k-1)x-\frac{1}{4}(k+1)^2\tau}u(x,\tau)$, where

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} \qquad for - \infty < x < \infty, \tau > 0 \tag{5.3}$$

with modified initial condition

$$u(x,0) = max(e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}, 0)$$

Boundary condition: The space direction is the asset price. Theoretical boundary values for the asset price are $-\infty$ and $+\infty$. When the price $x \longrightarrow -\infty$ at any time τ, we expect that the option contract has almost no value. The option value is supposed to be a discounted value. Here $u(x,\tau)$ is the final option value. That is,

$$\lim_{x\longrightarrow -\infty} u(x,\tau) = e^{\frac{1}{2}(k_1-1)x+\frac{1}{4}(k_1-1)^2\tau}$$

$$\lim_{x\longrightarrow \infty} u(x,\tau) = 0$$

The purpose of such a transformation is to make the application of numerical techniques easier.

5.2 Forward-Time Central-Space (FTCS) Algorithm

To solve equation 5.3, it needs to be discretized and rendered in algebraic form. We use the FTCS scheme to discretize the equation. The approach is as follows: A finite number of equally spaced time steps between the current date ($t = 0$) and the maturity date of the option, $(t = T)$ are chosen. Similarly, a finite number of equally spaced asset prices (N_j) are also chosen.

$$\Delta t = T/N,\ (N+1) \text{ total time steps}$$

$$\Delta S = S_{max}/2N_j,\ (2N_j + 1) \text{ total asset prices}$$

By the above discretization, a grid consisting of a total of $(N+1)(2N_j+1)$ points is constructed as shown in figure 5.1. The grid point (i, j) corresponds to time $i\Delta t$ and price $j\Delta S$. The third-dimension is the computational time step (note that this time step is different from the physical time step that marches from expiration to the current date) which is used to ensure the stability of the results. The accuracy of the original system is dictated by the convective term $(\frac{\partial u}{\partial S})$ rather than the diffusive term $(\frac{\partial^2 u}{\partial S^2})$, in the original non-transformed B-S equation. The "convective term" brings in the new information from the outside world into the trading place and convects into the system. Arrival of such information is mixed into the system by the mixing term or "diffusion term". Unless there is information to mix, the diffusion term loses its importance. Therefore, central-differencing for the convective term (2^{nd} order accurate) and backward or forward-differencing for the diffusion term (1^{st} order accurate) would be sufficient. However, we applied central-differencing to the diffusion term as well, in the transformed B-S equation because we felt that the additional computational cost incurred due to central-differencing to the diffusion term could be recovered with the multithreaded implementation of the algorithm.

The implementation details of the algorithm are described below with reference to Figure 5.1. The solution scheme is iterated over many *computational* time steps until it reaches a steady state. A steady state is defined as a scenario when the solution changes

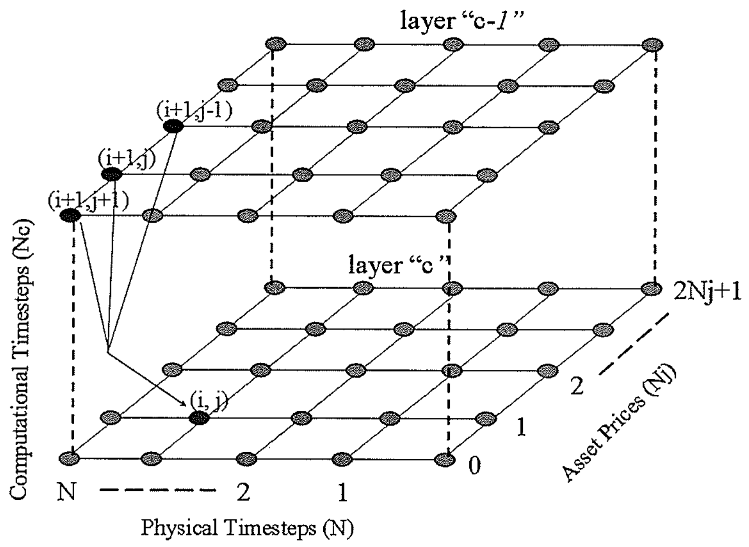

Figure 5.1: A three-dimensional mesh for computing option value with FTCS method.

very little between two consecutive computational time steps. Figure 5.1 shows an instance during the computation of the option price. The computed values of the horizontal layer $(c-1)$ are used to calculate the values of layer c. The number of computational layers depends on the relative error. The value $u_{i,j}^{c}$ denotes the option value at the (i,j) grid point in c^{th} computational time layer. The option value at the c^{th} layer is calculated as:

$$u_{i,j}^{c} = \rho u_{i+1,j-1}^{c-1} + (1-2\rho)u_{i+1,j}^{c-1} + \rho u_{i+1,j+1}^{c-1}$$

$\rho = l/h^2$, where l denotes the step size in the time direction τ and h denotes the step size in the price direction x. If τ ranges from 0 to T and x ranges from x_{min} to x_{max}, then $N \times l = T$ and $2N_j \times h = x_{min} - x_{max}$. Therefore, note that, the terminal condition $t = T$ is reformed to the initial condition $\tau = 0$ [TZG04].

The relative error is calculated as $err = u_{i,j}^{c} - u_{i,j}^{c-1}$. The computation steps once the err falls below a certain preset threshold value. The threshold for our algorithm is 10^{-4}, which gives us the accuracy of 1% of cent. Since the implementation uses an explicit scheme, by partitioning a given computational layer among many processors, we can achieve parallelism in computation and the computation of the option value in a particular layer is dependent only on the values in the previous layer (i.e. the previous computational time step). For a fundamental treatment on discretization, please refer to [ATP84, TR00].

5.3 Summary

Summarizing, we see the forward-differencing in time and central-differencing in space is best for the current problem to insure stability. Since there might be millions of nodes in each layer, only two layers are saved in the memory so as to use the memory resources efficiently (since the nodes in the current layer are computed only with the help of nodes in the preceeding layer).

The clients in this framework will be thin-clients (a simple client program which does not do any processing and relies on most of the function of the system being in the server) and all the processing will be done on the server-side. The compute server is also multithreaded and uses one thread-per-client policy to spawn threads using Java threads.

The next Chapter compares the speedup achieved using the various scheduling techniques for the server implementation. We also present our results of using different distributed clients over the network.

Chapter 6

Results

In this Chapter, we present the reults of our multithreaded algorithm. We start with a brief description of the factors that affect the option value of a contract and then follow it with our experimental performance results.

6.1 Factors Affecting the Option Value

When a client sends a request through CORBA for the computation of an option value, it passes three parameters that are used by the server to calculate the value of the option contract. The three parameters, as seen in Chapter 4, are: the expiration time (denoted by T), the strike price (denoted by E), and the number of levels (denoted by N). Other parameters, that are required for the option value computation are interest rate (r), volatility (σ), number of asset prices considered (N_j), and the stock price (S) of the option. For testing, these parameters are set to values shown in table 6.1 except when they were varied to study their effect on the option price.

Consider the effect of strike price E on the option value. Since the payoff in the case

Table 6.1: The initial values of various parameters.

S	\$100
E	\$100
r	10%
σ	10%
T	1 year

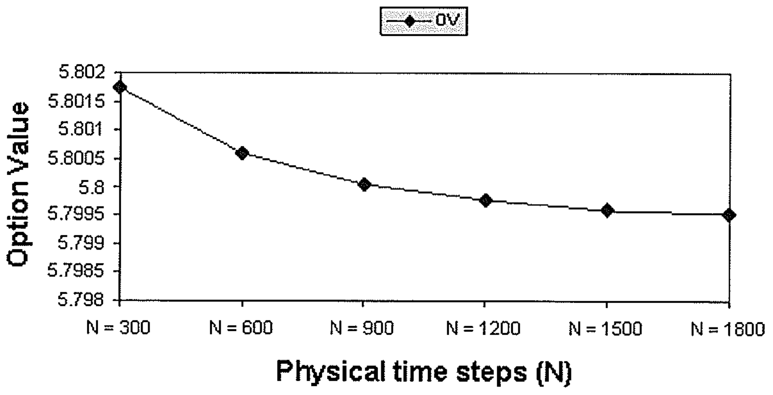

Figure 6.1: Variance with physical time steps (N).

of call option is $max(S - E, 0)$, the increase in E decreases the value of the option. That is, the option moves out-of-the-money. Put options on the other hand behave in the opposite manner, the value of the put option increases with the increase in the value of E. Thus, when $S \rightarrow \infty$, the value of a call option becomes S. Figure 6.1 shows the effect of increasing the number of levels (N) or in other words, dividing the total time into smaller time steps. As N increases, the option value converges to a stable value and thereby more accurate results are obtained.

Option value is directly proportional to volatility. The volatility of a stock price is a measure of how uncertain we are about future stock price movements. As the volatility increases, the probability that the stock will do very well or very poorly also increases. The holder of the call option benefits from price increases but has limited risk if the price decreases because the most the holder can lose is the premium (in our case we assume that there is no premium) that the holder paid for the option to the writer. Similarly, the holder of the put option benefits from price decreases, but has limited risk if the price increases. Both put options and call options become more valuable as the time to expiration increases because the holder of an option with longer life has more exercise opportunities than the holder of an option with shorter life.

The effect of risk-free interest rate (r) is not as clear as that of other factors. As interest rates in the economy increase, the expected return required by the investors from the stock tends to increase. Also, the present value of any future cash flow received by the holder of the option decreases. The combined result of these two effects is to decrease the value of put options and increase the value of call options. In practice, however, when the interest rate increases, the price of the stock tends to fall.

6.2 Experimental Results

In this section, we report our experimental results obtained in the implementation of the FTCS algorithm by varying the problem size and the number of threads on an eight processor SMP. All the processors are Intel Pentium III with 700MHz CPU and cache size of 1024KB. The total physical memory of the machine is over 7GB.

The clients used for the experiments are a network of computers which consisted of twenty-four heterogenous computers (8 machines each of Windows, Linux and Solaris) available in the Department of Computer Science. The tests were performed on all three types of clients which differed in operating systems (OS), CPU speed and memory. Table 6.2 lists the specifications of the clients.

Table 6.2: Specifiaction of different clients.

Clients	OS	CPU Speed	RAM
Windows	Windows XP	Intel pentium III 866MHz	256MB
Linux	Fedora Core 2.0	Intel pentium III 866MHz	256MB
Solaris	SunOS 5.8	UltraSPARC III 750MHz	512MB

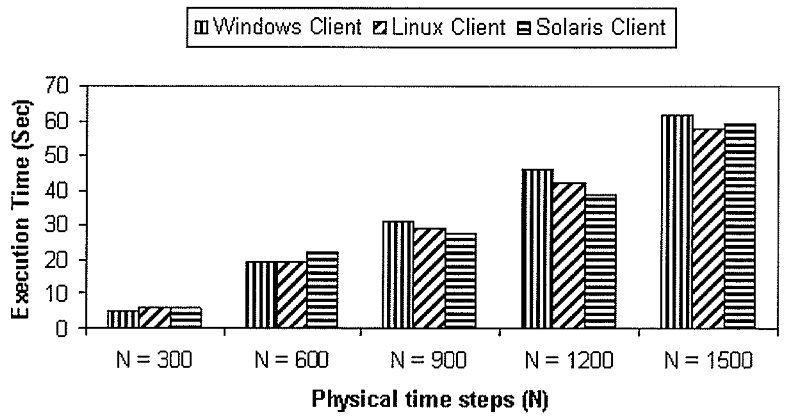

Figure 6.2: Execution time on different clients.

The clients used for the experiments are all dedicated machines (though the server was not). The execution times seen by the clients is the time taken by the compute server to calculate the option price using the multithreaded algorithm and includes the time taken to get the information from the remote database. The remote database of stock information has been implemented using DB2 and SQL servers running on *Neptunium* and *Uranium* servers in the Department of Computer Science. Figure 6.2 shows a comparison of execution time of different clients at $N_j = 200$ and with varying N (physical timesteps). Theoretically the execution time for all the clients should be approximately the same (irrespective of the OS and other specifications) as the clients are thin clients and do not do any significant processing by themselves. The slight variation in the execution time observed is likely due to the varying load on the compute server (the SMP machine) as it was not dedicated during testing.

Table 6.3: Option Value (OV) at various grid sizes with $N_j = 200$

N	**OV**	**Difference**
N=300	5.801736	-
N=600	5.800581	0.001155
N=900	5.800064	0.000517
N=1200	5.799780	0.000284
N=1500	5.799583	0.000197
N=1800	5.799524	0.000059

Table 6.3 shows option value (OV) with $N_j = 200$ for various N's. Over a large number of compute time steps it is expected that the option value should converge, which is very clear from this table.

The computation of the option value at an intermediate node of a layer (C_i) depends on the values of the previous layer (C_{i-1}). The nodes in C_i are divided among different threads to speedup the computation process. In this work, we use Java OpenMP (JOMP) [BWKO00, EPC, Obd99], which provides a collection of compiler directives, library functions, and environment variables that can be used for shared memory parallel programming in Java. To use the JOMP directives and functions, a file with *.jomp* extension is created which is then converted to the corresponding *.java* file with the help of the JOMP compiler [BWKO00]. The JOMP compiler basically replaces the parallel constructs in the *.jomp* file with a new instance of a compiler created class and the fields within it are initialized to appropriate variables. The 3-D grid mesh used for the computation is constructed with the help of multiple *for* loops, therefore, only the "for" work-sharing construct of JOMP and its clauses are discussed here. For more details on other directives and constructs, please refer to [BWKO00, Obd99].

The general form [Obd99] of a parallel for in JOMP is given by:

$$//ompfor[clause[clause[\cdots]$$

$$for - loop$$

There are many different possible clauses. We used the *private*, *schedule* and *nowait* clauses in our experiments. The *nowait* clause is used to remove the implicit barrier at the end of the loop to attain maximum speedup. It is safe to the *nowait* clause in our case as we are parallelizing individual layers and the computation of nodes in a specific layer deos not depend on any other node in the same layer. The *private* clause declares the variables in the associated list to be private to each thread in a team. The *schedule* clause specifies how the iterations of a loop are to be divided among the threads. The value of the chunk size (*CS*) variable, if specified, determines the number of iterations assigned to one thread. The schedule clause can be set to one of the following:

static: When *schedule(static, CS)* is specified, the iterations of the loop are divided into chunks of size *CS*. The value of *CS* has to be loop-invariant and positive. The chunks are then statically assigned to threads in a round-robin fashion based on the thread number. If no chunk size is specified, then the iterations are divided in approximately equal chunks that will minimize the total number of iterations performed by any thread.

dynamic: When *schedule(dynamic, CS)* is specified, a chunk of *CS* iterations is assigned to each thread. On completion, the thread is assigned the next available chunk dynamically (i.e. at runtime). If the value of CS is not specified, the default value is 1.

guided: When *schedule(guided, CS)* is specified, the iterations are also assigned dynamically, but the chunks have exponentially decreasing sizes. If the value of CS is not specified, the default is 1.

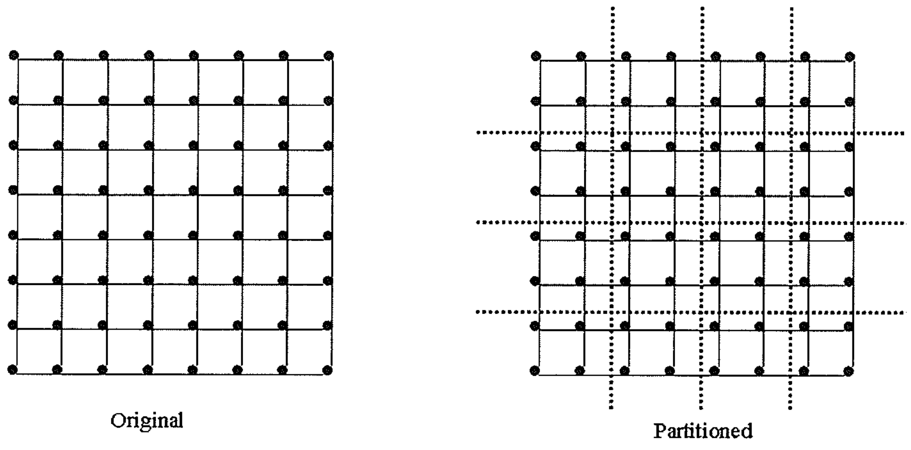

Figure 6.3: Original and partitioned grid that uses various scheduling techniques.

Figure 6.3 shows an instance of a 2-D layer of asset prices and physical time before and after partitioning using the static scheduling techniques. The grid shown here is symmetric and therefore, the CS for all the threads is equal. The last chunk may be of smaller size depending on the size of the grid and the number of threads used. However, in the guided technique, the CS decreases exponentially and the subsequent chunks are approximately the number of remaining iterations divided by twice the number of threads. We first present speedup results obtained using the default scheduling technique and later we show the effect of using other scheduling techniques. Speedup is the ratio of execution time of a sequential program to that of multi-threaded parallel program.

Figures 6.4 and 6.5 show the execution time and speedup obtained with 2-, 4-, 8-, and 16-threads on 8 processors respectively with 200 asset values ($N_j = 200$). We achieved a speedup ranging between 1.60 to 3.99, the maximum being 3.99 using 16-threads (on

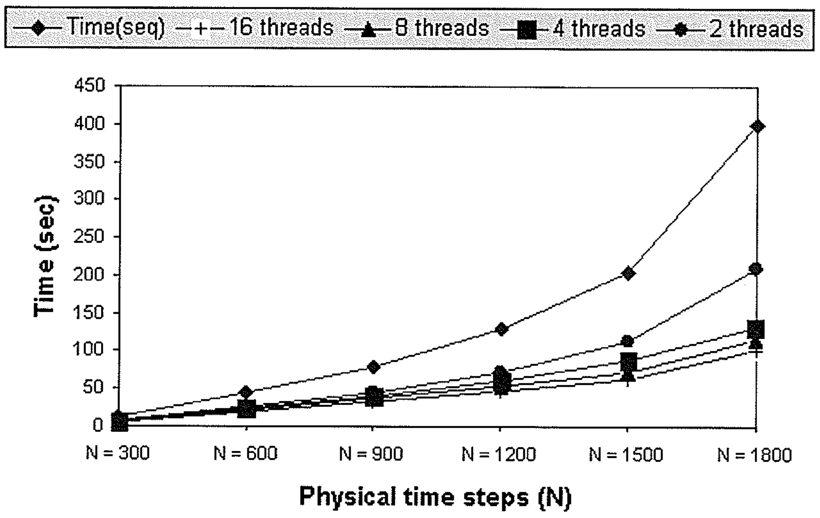

Figure 6.4: Execution Time (in seconds) with varying time steps and 200 asset values.

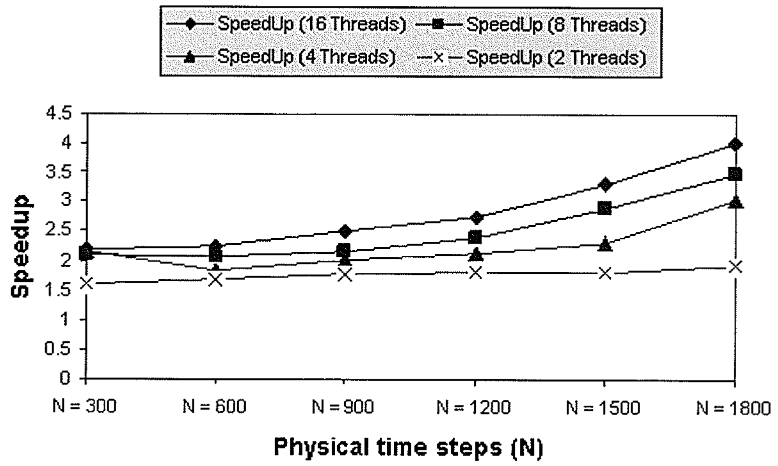

Figure 6.5: Speedup with varying time steps and 200 asset values.

8 processors) with 1800 physical time steps (N). Here we used the default scheduling technique of JOMP. Threads in all cases are evenly distributed among processors. With 16-threads (2 threads per processor) we get the most speedup. Other experiments show that, on further increasing the number of threads, we did not get any more speedup due to the context switching and synchronization delays.

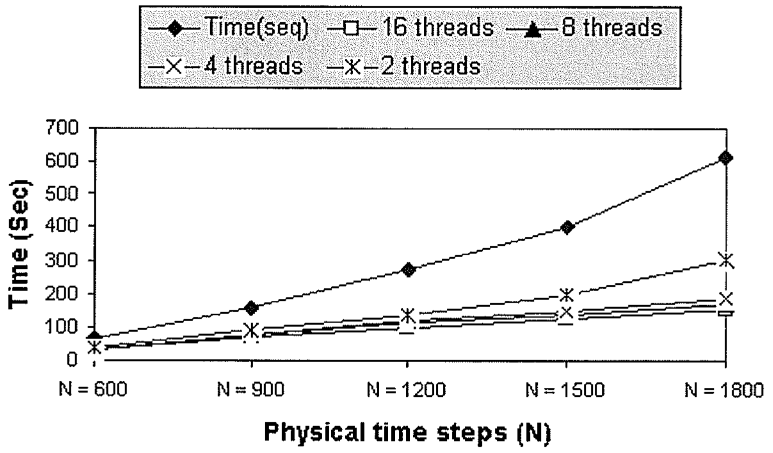

Figure 6.6: Execution Time (in seconds) with varying time steps and 400 asset values.

Figures 6.6 and 6.7 show the execution time and speedup results obtained after increasing the number of asset values from 200 to 400. Notice the increase in speedup with the increase in the number of threads and the problem size. The reason for dividing the expiration time into a maximum of 1800 small time steps is because there are approximately 225 working days in a year with 8 working hours every day. Hence, we take N to the maximum of $1800(225 * 8)$ which gives hourly precision of the option value.

Since the best results were obtained using 16-threads and $N_j = 200$, we keep $N_j = 200$ and use 16-threads to test the different scheduling techniques in JOMP. We have experimented with all three scheduling techniques provided in JOMP and Figure 6.8 shows the speedup results. In these experiments, *CS* was $\frac{N}{16}$ for all scheduling techniques and

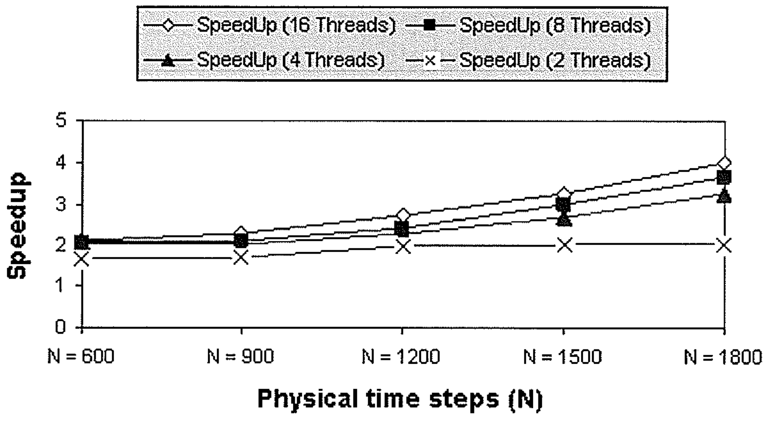

Figure 6.7: Speedup with varying time steps and 400 asset values.

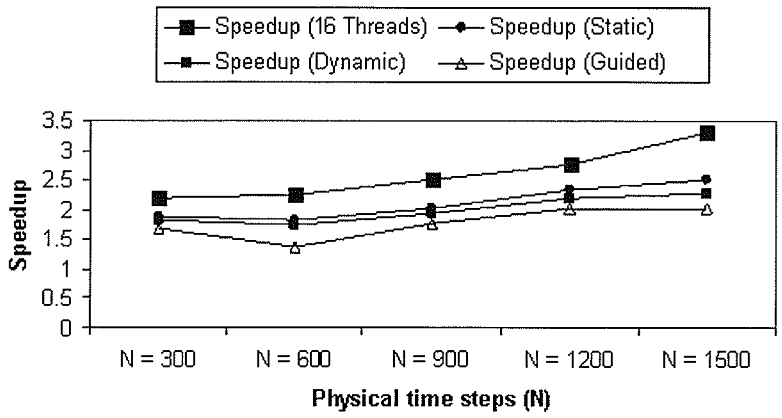

Figure 6.8: Speedup obtained with four different scheduling techniques.

results labelled “Speedup (16 Threads)” correspond to the default scheduling technique. Notice that the default scheduling scheme yields the best results, this is because JOMP divides the iterations so as to minimize the maximum number of iterations per thread. This scheme reduces the context switching time of the threads. In static and dynamic scheduling, each thread is given an equal amount of work initially. Therefore, the two scheduling policies do not show much difference. The speedup with dynamic scheduling is slightly less than static as some threads under the dynamic scheduling scheme execute the last few chunks while the others sit idle. Hence, this load imbalance causes the dynamic scheme to be slightly slow. The guided scheme, however, gave the least speedup because the specified chunk size is $\frac{N}{16}$ to begin with and then the chunk size decreases exponentially [KOB01, BWKO00] and hence more threads are required to do the same computation at the end. More threads cause synchronization and context switching delays which affects the performance of the algorithm.

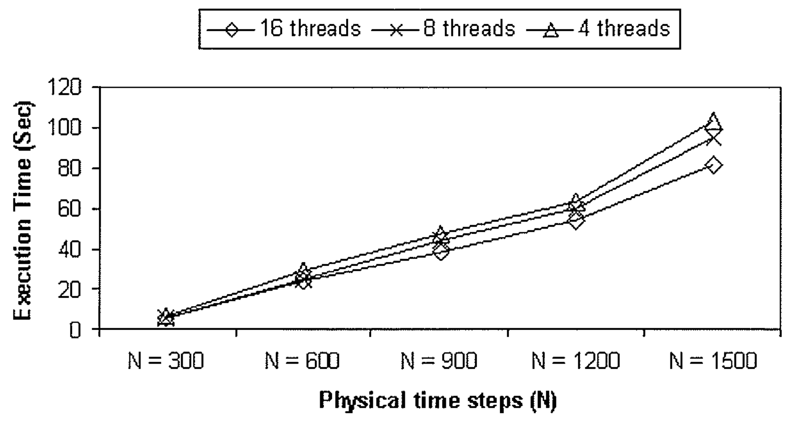

Figure 6.9: Execution time: Keeping the CS constant and varying the number of threads.

Figure 6.9 shows the result of keeping the *CS* constant at $\frac{N}{16}$ and varying the number of threads. We do not notice much difference in the execution time until $N = 900$. However, on further increasing the number of physical time steps N, 16-threads give the best results. As N increases, the number of chunks to be executed by the threads also increases. For example, if N is 1500, there are 94 chunks to be executed by the threads. In the case of 16-threads, 14-threads execute 6 chunks and 2-threads execute 5 chunks each. Whereas, with 8-threads, 6-threads execute 12 chunks and 2-threads execute 11 chunks each. Therefore, there is more context switching overhead in case of lower number of threads. Figure 6.10 shows the execution time of our algorithm on varying CS with 16-threads using the static scheduling technique. Here, the equal chunks are statically assigned to each thread and we notice that we get best results when CS is equal to the number of threads.

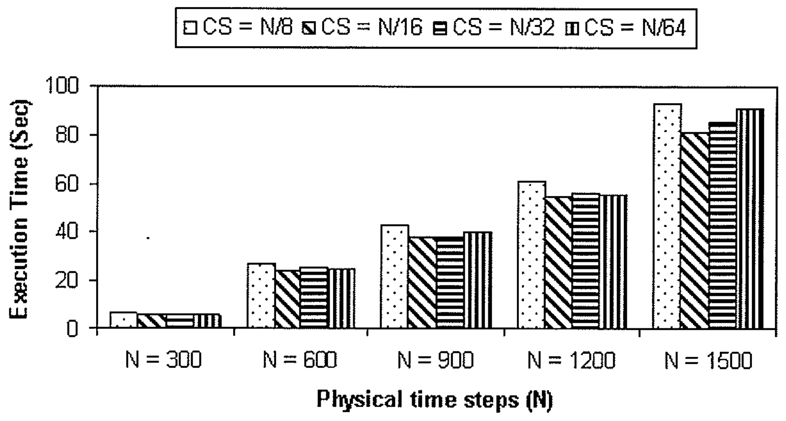

Figure 6.10: Execution time: Keeping the number of threads constant and varying CS.

6.3 Summary

In summary, the default scheduling technique of JOMP gave the best results. Out of the other three scheduling techniques, the static technique gave best results with CS $= \frac{N}{16}$. 16-threads gave the maximum speedup of 3.99 on eight processors and on further increasing the number of threads, the increase in context switching and synchronization delays decreased the speedup.

Chapter 7

Conclusions and Future Work

This thesis presented a framework for evaluating Eurpoean options. There are three contributions in this work: (i) the development of a multithreaded algorithm to solve the B-S model for option pricing problem. (ii) the use of JOMP for creating a computing environment for option pricing; (iii) integrating a CORBA based client and back end server (that computes the option values using *multithreaded* option pricing algorithm using FTCS finite-difference technique).

We tested our algorithm on a heterogeneous with nodes varying in OS, CPU speed and memory capacity. We analyzed the parallel run times of the approach and presented the speedup results. We experimented with our algorithm for various chunk sizes and using three different scheduling techniques (static, dynamic, guided) provided in JOMP. We studied the behavior of static scheduling by varying the number of threads and chunk sizes. Out of the three scheduling policies, the static policy outperformed the other two scheduling techniques. The maximum speedup obtained on an SMP machine with eight processors was approximately 4 using 16-threads.

We also showed that the method is scalable. That is, it can maintain constant parallel efficiency if the number of threads and the size of the problem are increased correspondingly. We have validated the accuracy of our algorithm for pricing European call option on one asset. We showed that the option value becomes stable as the problem size is increased (in other words, having more intermediate points in a computation for a given period of the option contract) and divergence between the option values decreases with a smaller number of computational time steps.

We presented the parallel implementation of a 3-dimensional mesh for the evaluation of European options using a database of stock information. We reduced the B-S equation to a form (equation 5.3) which made it easier to implement. We used JOMP to parallelize the FTCS finite-difference method on shared memory architecture. Thus, in rapidly changing market place, the results can help the investor to make informed decisions.

In future, we would like to test our algorithm on a SMP with more than eight processors, such as the 24 processor *SunFire* system at the University of Manitoba. Other possible extensions that can come out of this thesis are:

- To incorporate various Real-Time CORBA (RT-CORBA) [SK00] which would allow us to configure and control the processor, communication, and memory resources.
- Study the problem with other explicit and implicit finite-difference schemes for solving the Black-Scholes equation. Compare the results from different techniques for accuracy and speedup.
- Since FLEET is a client-server framework, we can extend the framework to include a multitude of services to the investor (for example, portfolio optimization).

Bibliography

[ACC$^+$90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of International Conference of Supercomputing*, pages 1–6, Amsterdam, Netherlands, June 1990.

[AI87] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of Parallel Processing on Science and Engineering*, University Press, 1987.

[ATP84] D. A. Anderson, J. C. Tannehill, and R. H. Pletcher. *Computational fluid dynamics and heat transfer*. Hemisphere Publishing Corporation, New York, USA, 1984.

[AWDN04] A.D. Andricopoulos, M. Widdicks, P.W. Duck, and D.P. Newton. Curtailing the range of lattice and grid methods. *Journal of Derivatives*, 11(4):55–61, June 2004.

[Bar04] S. Barua. Fast Fourier transform for option pricing: Improved mathematical modeling and design of an efficient parallel algorithm, Department of Computer Science. Master's thesis, University of Manitoba, Winnipeg, MB, Canada, July 2004.

[Bat80] K. E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, September 1980.

[BBK$^+$68] G.H. Barnes, N.M. Brown, M. Kato, D. Kuck, D. Slotnick, and N.A. Stokes. The ILLIAC-IV Computer. *IEEE Transactions on Computers*, 17(8):746–757, 1968.

[BC97] G. Bakshi and Z. Chen. An Alternative Valuation Model for Contingent Claims. *Journal of Financial Economics*, 44(1):123–165, 1997.

[BJK$^+$95] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Run Time System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, California, July 1995.

[Bol02] F. Bolton. *Pure CORBA*. SAMS Publishing Company, Indianapolis, Indiana, USA, 2002.

[BS73] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–654, January 1973.

[BS77] Michael J. Brennan and Eduardo S. Schwartz. The valuation of American put options. *The Journal of Finance*, 32(2):449–462, May 1977.

[BTT04] S. Barua, R. K. Thulasiram, and P. Thulasiraman. Fast Fourier transform for option pricing: Improved mathematical modeling and design of an efficient parallel algorithm. In *Proceedings of the International Conference on Computational Science and its Applications (ICCSA04)- Vol.3*, pages 686–696, Assissi, Italy, May 2004. Springer-Verlag Lecture Notes in Computer Science.

[BWKO00] J. M. Bull, M. D. Westhead, M. E. Kambites, and J. Obdrzalek. Towards OpenMP for Java. In *Proceedings of the Second European Workshop on OpenMP (EWOMP)*, Edinburgh, Scotland, U.K., September 2000.

[Cer04] A. Cerny. Introduction of fast Fourier transforms in finance. *Journal of Derivatives*, 12(1):73–88, October 2004.

[Cla98] I. J. Clark. Option pricing algorithms for the Cray T3D supercomputer. *Proceedings of the first National Conference on Computational and Quantitative Finance (loose bound volume, no page numbers)*, September 1998.

[CLV02] T. Coleman, Y. Li, and A. Verma. A newton method for american option pricing. *The Journal of Computational Finance*, 5(3):51–78, Spring 2002.

[CM99] P. Carr and D. B. Madan. Option valuation using the fast Fourier transform. *The Journal of Computational Finance*, 2(4):61–73, 1999.

[Cor87] NCUBE Corporation. Ncube user handbook, 1987.

[CRR79] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, September 1979.

[CTT04] A. Chhabra, P. Thulasiraman, and R. K. Thulasiram. Distributed computing with CORBA and multithreading with Java OpenMP on NoCs: A framework for CFD. In *Proceedings of the twelfth International Conference on Advanced Computing and Communication (ADCOM)*, pages 489–495, Ahmedabad, Gujarat, India, Dec 15-18 2004.

[Den91] J. B. Dennis. *The Evolution of 'Static' Data-Flow Architecutre*, chapter 2. Prentice-Hall, Englewood Cliffs, New Jersy, USA, 1991. In Advanced Topics in Dataflow Computing, Editors: J-L Gaudiot and L. Bic.

[EPC] EPCC. Java OpenMP. http://www.epcc.ed.ac.uk/research/jomp/.

[FN04] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault tolerant CORBA systems. *IEEE Transactions on Computers*, 53(8):497–511, 2004.

[GBT87] J.R. Gurd, A.P.W. Bohm, and Y.M. Teo. Performance issues in dataflow machines. *Future Generation of Computer Systems*, 3:285–297, 1987.

[GGKK03] A. Grama, A. Gupta, V. Kumar, and G. Karypis. *Introduction to Parallel Computing*. Pearson Educarion Limited, Edinburgh Gate, Essex, 2 edition, 2003.

[GKB87] J.R. Gurd, C.C. Kirkham, and A.P.W. Bohm. *The Manchester Dataflow Computing System*, pages 177–219. North Holland Publishing Company, 1987. In Experimental Parallel Computing Architectures, Editor: J. J. Dongarra.

[Gro97] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP). http://www.omg.org/cgi-bin/doc?formal/97-02-05.pdf, February 1997. (OMG).

[Hes93] S. L. Heston. A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *Review of Financial Studies*, 6:327–343, 1993.

[HMT$^+$95] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A design study of the earth multiprocessor. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 1995. ACM Press.

[Hof89] K. A. Hoffmann. *Computaional Fluid Dynamics for Engineers*. Engineering Education System, Austin, Texas, USA, 1989.

[HT05] K. Huang and R. K. Thulasiram. Parallel algorithm for pricing american asian options with multi-dimensional assets. In *Proc. (CD-RoM) 19th Intl. Symp. High Performance Computing Systems and Applications (HPCS)*, Guelph, ON, Canada, May 2005. (To appear).

[Hul02] J. C. Hull. *Options, futures and other derivatives*. Prentice Hall, Upper Saddle River, NJ, 5th edition, 2002.

[Inc] Cray Inc. Cray history. http://www.cray.com/about-cray/history.html. An article describing the evolution of Cray.

[JLL90] P. Jaillet, D. Lamberton, and B. Lapeyre. Variational inequalities and the pricing of American options. *Acta Applicandae Mathematica*, 21:263–289, 1990.

[KNR00] E. J. Kontoghiorghes, A. Nagurnec, and B. Rustem. Parallel computing in economics, finance and decision-making. *Parallel Computing*, 26:207–509, 2000.

[KOB01] M. E. Kambites, J. Obdrzalek, and J. M. Bull. An OpenMP-like interface for parallel programming in Java. *Concurrency and Computation: Practice and Experience*, 13(8-9):793–814, 2001.

[May00] A. Mayo. Fourth order accurate implicit finite-difference method for evaluating American options. In *Proceedings (CD-RoM) of the International Conference on Computational Finance 2000*, London, England, June 2000.

[Obd99] J. Obdrzalek. OpemMP for java. http://www.epcc.ed.ac.uk/research/jomp/papers/ssp0599.pdf, 1999. Technical Report.

[Org] OpenMp Organization. Openmp. http://www.openmp.org/drupal/. The homepage for the OpenMP Organization.

[OW99] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Cambridge, MA, 1999.

[Pub03] SGI Technical Publications. Sgi - c language reference manual, June 2003.

[R. 01] R. Chandra and L. Dagum and D. Kohr and D. Maydan and J. McDonald and R. Menon. *Parallel Programming in OpenMP*. Moran Kaufmann, San Francisco, CA, 2001.

[RST04] S. Rahmail, I. Shiller, and R. K. Thulasiram. Different estimators of the underlying asset's volatility and option pricing errors: Parallel Monte Carlo simulation. In *Proc. International Conference on Computational Finance and its Applications*, pages 121–131, Bologna, Italy, April 2004.

[RST05] S. Rahmail, I. Shiller, and R. K. Thulasiram. Cost of option pricing errors associated with incorrect estimates of the underlying assets volatility: Monte carlo simulation. In *Proceedings of 11th IMACS International Conference on Monte Carlo Methods*, Tallahassee, FL, USA, May 2005.

[Sco97] L.O. Scott. Pricing Stock Options in a Jump-Diffusion Model with Stochastic Volatility and Interest Rates. *Mathematical Finance*, 7(4):413–426, 1997.

[Sei85] C. L. Seitz. The cosmic cube. *Communications of ACM*, 28(1):22–33, 1985.

[SGR99] D. Slama, J. Garbis, and P. Russel. *Enterprise CORBA*. Prentice Hall, New Jersy, USA, 1999.

[SK00] D. C. Schmidt and F. Kuhns. An overview of Real-time CORBA Specification. *IEEE Computer Magazine*, 33(6):55–63, June 2000.

[Sok96] A. D. Sokal. Monte carlo methods in statistical mechanics: Foundations and new algorithms, February 1996. Lecture Notes, Department of Physics, New York University.

[Sri02] A. Srinivasan. Parallel and distributed computing issues in pricing financial derivatives through quasi monte carlo. In *Proc. (CD-RoM) International Parallel and Distributed Processing Symposium (IPDPS02)*, Fort Lauderdale, FL, April 2002.

[TB91] Y.M. Teo and A.P.W. Bohm. *Resource Management in Data-Flow Computers with Iterative Instructions*, pages 481–499. Prentice-Hall, Englewood Cliffs, New Jersy, USA, 1991. In Advanced Topics in Dataflow Computing, Editors: J-L Gaudiot and L. Bic.

[TB02] R. K. Thulasiram and D. Bondarenko. Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. In *IEEE Computer Society Proceedings of the Fourth International Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA 2002)*, pages 306–313, Vancouver, BC, Canada, August 2002.

[TB05] R. K. Thulasiram and D. Bondarenko. Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. *International Journal of Computational Science and Engineering (accepted)*, 2005. (To appear).

[TLN$^+$01] R. K. Thulasiram, L. Litov, H. Nojumi, C. T. Downing, and G. R. Gao. Multithreaded algorithms for pricing a class of complex options. In *Proceedings (CD-ROM) of the International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, April 2001.

[TR00] D. Tavalla and C. Randall. *Pricing financial instruments: The finite difference method.* John Wiley and Sons, New York, NY, 2000.

[TT03] R. K. Thulasiram and P. Thulasiraman. Performance evaluation of a multithreaded fast Fourier transform algorithm for derivative pricing. *The Journal of Supercomputing*, 26(1):43–58, August 2003.

[TZG04] R. K. Thulasiram, C. Zhen, and A. Gumel. A second order L_0 stable algorithm for evaluating European options. In *Proc. 18th International Symposium on High Performance Computing Systems and Applications*, pages 17–23, Winnipeg, MB, Canada, May 2004.

[WHD95] P. Wilmott, S. Howison, and J. Dewynne. *The mathematics of financial derivatives.* Cambridge University Press, Cambridge, UK, September 1995.

[Zen99] S. A. Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25:2149–2075, December 1999.

[ZFV98] R. Zvan, P. Forsyth, and K. R. Vetzal. A general finite element approach for PDE option pricing models. *Proceedings of the first National Conference on Computational and Quantitative Finance (loose bound volume, no page numbers)*, September 1998.