Distributed Quasi-Monte Carlo Algorithm for Option Pricing on HNOWs Using mpC

by

Gong Chen

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

Master of Science

Department of Computer Science

Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Distributed Quasi-Monte Carlo Algorithm for Option Pricing on HNOWs Using mpC

BY

Gong Chen

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

This thesis is dedicated to my parents and brothers.

Abstract

Monte Carlo (MC) simulation is one of the popular approaches for approximating the value of an option or other derivative security in addition to binomial lattice techniques. However, plain MC simulation produces only approximate solutions. Absence of straightforward closed form solutions for many financial models for pricing option has given rise to use of numerical approaches. The introduction of low-discrepancy (LD) sequences in MC simulation provides a way to improve the accuracy and reliability of MC methods. The use of LD sequences in MC method leads to what is known as Quasi-Monte Carlo (QMC) method. Several studies have investigated efficiency of such methods on serial computers. This research will focus on the parallelization of the QMC method on a heterogeneous network of workstations (HNOWs) for option pricing. HNOWs are machines with different processing capabilities and have distinct execution time for the same task. So it is very important to allocate and schedule the tasks depending on the performance and resources of these machines. Some of the existing parallelization of traditional MC approach use SPMD (Single Program Multiple Data) manager-worker paradigm and communication between the machines is by message passing using standard MPI (Message Passing Interface) library. On heterogeneous machines, MPI is not the appropriate programming library. It does not consider the underlying features of the machines while scheduling the tasks.

In this research, we have developed an adaptive, distributed QMC algorithm for option pricing, taking into account the performance of processors and communication latencies. On heterogeneous networks, performance gains are potentially available for

algorithms if they are designed to fully exploit the hardware features. This is the very peculiarity of our parallel algorithm, which takes into account the actual performances of both processors and communication links. We implemented the algorithm using mpC, an extension of ANSI C language for parallel computation on HNOWs. mpC addresses issues related to heterogeneous computing environments and is an ideal language for our problem. An outstanding feature of using mpC is that a programmer can specify the topology of the application under study and mpC system can map the topology to real network system based on processors' processing speeds and network bandwidths in run time. By comparing with other parallel algorithms and implementations, the speedups exhibited by our algorithm presented in this thesis are promising.

Acknowledgements

I would like to express my thanks to all those who gave me the possibility to complete this thesis.

First of all, I would like to express my sincere gratitude and appreciation to my supervisors Dr. Parimala Thulasiraman and Dr. Ruppa K. Thulasiram (Tulsi) for their constant advice, guidance, encouragement, and efforts in helping with preparing this thesis. Dr. Thulasiraman is an active Researcher in High Performance Computing and Algorithms; she introduced me the parallel programming language mpC and its programming environment. Dr. Tulsi is a reputable specialist in Computational Finance and Scientific Computing; he taught me the Quasi-Monte Carlo method for Option pricing. Both of them helped me to establish the overall direction of the research and spent numerous hours in every phase of my research and patiently answered all my questions, and helped me to move forward progressively to the accomplishment of the M.Sc program. This thesis would have not been possible without them.

I would like to thank my thesis committee members, Dr. Ben Pak-Ching Li (internal) and Dr. Saumen Mandal (external), for managing to read the whole thing so thoroughly and attending my defense. Thank-you to Dr. Dean Jin for the time in serving as the Chair of my thesis defense.

I am grateful to Mr. Gilbert Detillieux for helping to install the mpC programming system. Without his help, my algorithm cannot be implemented.

I am thankful to Mr. Santan Challa who helped me to run the first mpC program "Hello, world!". Without this initial help, I cannot run my mpC program. I have also

benefited from the discussions with him about mpC programming.

I would also like to thank many developers in mpC system team, especially, Dr. Mikhail Posypkin and Dr. Alexey Kalinov, for their remote help in configuring the mpC programming environment and answering questions.

I also thank all members of the Parallel Algorithm Research In Manitoba's ALgorithms and Application (PARIMALA) Lab for all kinds of help.

Thanks to all my friends for their help and the days we spent together.

Finally, I am forever indebted to my parents and brothers for their understanding, endless patience, support, and encouragement. This thesis is dedicated to them.

Contents

1 Introduction 1

1.1 Goal of the Thesis 3

1.2 Organization of the Thesis 4

2 Parallel and distributed computing 5

2.1 Classification of Architectures 6

2.1.1 Single Instruction Multiple Data 6

2.1.2 Multiple Instruction Multiple Data 7

2.2 Message Passing Library 9

2.2.1 Parallel Virtual Machine 10

2.2.2 Message Passing Interface 11

2.2.3 Advantages and Disadvantage of MPI and PVM 12

2.3 mpC 13

2.4 Methodology for Creating Parallel Programs 15

3 Monte Carlo and Quasi-Monte Carlo Methods 17

3.1 Monte Carlo Method 17

3.2 Quasi-Monte Carlo Method 19

3.3 Sobol Sequence 22

3.4 Generating Sobol's LD Sequence 24

4 Option pricing and related work **26**

4.1 Definitions 26

4.2 Option Pricing 27

4.3 Related Work 28

4.3.1 Binomial Trees 28

4.3.2 Finite Difference 31

4.3.3 Monte Carlo Simulation 33

4.3.4 Quasi MC Simulation 34

5 Quasi-Monte Carlo Method for Option Pricing **35**

5.1 Monte Carlo Method 35

5.2 A Numerical Example 38

5.3 Quasi-Monte Carlo Method 40

6 Parallel Quasi-Monte Carlo Method for Option Pricing **43**

6.1 Tasks Partition 45

6.2 mpC 46

6.3 Implementation Detail 48

7 Results and Evaluation **55**

7.1 Analytical Results 55

7.2 Experimental Results 57

8 Conclusions and future work **67**

References **68**

List of Tables

5.1 A spreadsheet for estimating the expected present value of the payoff of a European call option 38

5.2 Numerical example for Monte Carlo valuation of a European Call option 41

7.1 Relative performance of 7 heterogeneous workstations 59

7.2 Distribution of 1,000,000 simulations to 7 heterogeneous workstations . . 60

7.3 Time to do QMC simulations in seconds 60

7.4 Time to do 1,000,000 simulations using three schemes 64

List of Figures

2.1 The SIMD architecture 7
2.2 The MIMD architecture 8
2.3 Shared memory machines and distributed memory machines 8

3.1 Plot of 500 Paris of Random Numbers 18
3.2 The first 16 number of sequence distributed over the interval $[0, 1)$ 20
3.3 Plot of 500 Paris of Sobol LD Numbers 21

4.1 Payoff function of the call option 28
4.2 A four-step Binomial tree for an asset 29
4.3 Grid for finite difference approach 32

5.1 Simulated asset price 39
5.2 Relative pricing error for a European Call option using pseudo-random numbers and Sobol LD numbers 42

7.1 Execution time with respect to various processors and simulations 61
7.2 Speedups computed relative to sequential code running on workstation Cadmium01 62
7.3 Manager-worker scheme with different amount tasks per request 65

Chapter 1

Introduction

In recent years, there has been increasing use of numerical methods in computational finance. This is due to the fact that most financial models lack straight-forward closed form solutions. There are three popular numerical methods used for option pricing, namely, the binomial lattice [21], the finite difference [14], and the Monte Carlo (MC) simulation [10]. In the real world, financial market is full of uncertainties in trends, price, etc. To obtain accurate results, a large number of state variables will be involved to imitate the real-world. Due to the complexity of these random factors involved, binomial lattice and finite difference methods become costly in terms of computational cost when three or more variables are involved [39]. In this case, MC simulation is a promising alternative method to evaluate options since the method is flexible and modern computing power ensures a quicker result, though MC simulations is known to provide less accurate solutions.

The MC method is a stochastic technique based on the use of random numbers and probabilistic methods to generate market conditions. The basic idea of using MC simulation to value option is to generate a large number of random configurations and evaluate the option value as the average of the sample [39]. The error in the MC estimation decreases at the order $O(N^{-1/2})$ where N is the number of simulations [31]. Hence, the estimation tends to the actual value as the simulations tend to infinity. Therefore, for

high order accuracy, large number of simulations are required. To improve efficiency and accuracy, mathematicians have found that using uniformly distributed deterministic numbers rather than random (or pseudo-random) numbers can obtain faster convergence with known error bounds. The error bounds in these methods are in the order of $(\log N)^d \cdot N^{-1}$ [51] where d is the problem dimension and N is the number of simulations. The uniformly distributed deterministic numbers is known as low discrepancy sequence (LD sequence). The use of LD sequences in MC method leads to what is known as Quasi-Monte Carlo (QMC) method. There are several methods for generating such LD sequences and these procedures are generally based on number theoretic methods. For a comprehensive survey of QMC methods, please refer to the monograph [51].

Due to the replicative nature, QMC simulation often consumes large amount of computing time. Solution on a sequential computer will require hours and may be even days depending on the size of the problem [56]. In financial markets, there is a high premium on rapid solution. Any rapid solution in information processing can be translated into potential gains. Therefore, parallel computing is an ideal choice since it provides a solution for large computational problems in a reasonable amount of time using more than one processing units. QMC simulations are well suited to parallel computing since it is an embarrassingly parallel problem (no communication between processors [61]). We can employ many processors to simulate various random walks and produce their values, then average these values to produce a final answer. Therefore, minimizing the whole simulation time.

Parallelization of QMC technique has gained importance in recent years and there's a growing trend towards using inexpensive workstations and PCs for parallel computing. These PCs, workstations, servers and sometimes supercomputers connected together form a heterogeneous network. Due to the varying processing capabilities of the processors, different operating systems and user load, these machines have different execution time for the same task and making the maximum benefits of the various processors is one of the important issues in heterogeneous networks. To ensure a successful and efficient

QMC simulation in such a distributed environment, it is very important to allocate and schedule the tasks depending on the performance and resources of these machines.

This research addresses issues on parallelizing QMC on a heterogeneous computing environment. Some of the existing parallelization approaches (e.g. a manager-worker paradigm) are implemented using standard MPI (Message Passing Interface) library. On heterogeneous machines, MPI is not the appropriate programming library [46]. It does not consider the underlying features of the machines while scheduling the tasks.

In this research, mpC is used to implement the QMC algorithm on heterogeneous network of computers. mpC is an extension of the ANSI C language for programming parallel computations on heterogeneous computers. mpC addresses issues related to heterogeneous computing environments and is an ideal tool for this problem. We will also compare and analyze the performance results on a homogeneous network using MPI.

1.1 Goal of the Thesis

The goal of this research is to develop and implement an adaptive, distributed QMC algorithm on heterogeneous network of workstations (HNOWs) using mpC. The term "adaptive" has two meanings: (1) this algorithm can be performed on any number of processors. That is, the general assumption that the number of processors is a factor of the number of simulations is relaxed; and (2) the tasks will be distributed to processors based on processors' processing capabilities. These runtime issues are not generally considered in previous studies of parallel QMC simulations. This is, in fact, the main goal of this project, as there has been relatively little done on HNOWs for the option pricing problem. Despite a growing need for efficient algorithms and implementations for option pricing problem, the parallel and distributed computing issues in option pricing are seldom addressed in the literature.

1.2 Organization of the Thesis

The rest of the thesis is organized as follows. The background on parallel and distributed computing is provided in Chapter 2. Monte Carlo and Quasi-Monte Carlo methods as well as concepts of low-discrepancy sequences are introduced in Chapter 3. The background and related work on computational finance which includes three popular numerical methods is presented in Chapter 4. Next, we address the sequential QMC algorithm in Chapter 5 followed by a detailed description of our parallel QMC algorithm and implementation details in Chapter 6. In Chapter 7, we show the analytical and performance results of our algorithm. Finally, we present our conclusions and future work in Chapter 8.

Chapter 2

Parallel and distributed computing

Parallel and distributed computing are widely used in a variety of areas ranging from academic to industry, such as computational simulations for scientific and engineering applications, commercial application in data mining and transaction processing. With the rapid enhancement of the bandwidth of interconnection networks and performance of PCs and workstations, as well as the continuous arrival of new languages and improvement of operation systems, parallel and distributed computing can achieve relatively very high performance with low cost for information processing; almost every application domain can profit.

Parallel computing is the simultaneous execution of a single task (split up and specially adapted) on multiple processors in order to obtain faster results, while distributed computing studies the coordinated use of physically distributed computers [47]. In the literature, these two terms are loosely used though the issues and research problems involved in these two paradigms are completely different. However, the common output in both cases is fast execution of an application. Performance of an application is therefore an important issue. In this chapter, we will review some of the architectures and relative programming models and environments for parallel computers.

2.1 Classification of Architectures

All computers, whether sequential or parallel, operate by executing instructions on data. Based on the number of streams of instructions performing on the data, Flynn [24] proposed a classification of computer architectures:

- Single Instruction Single Data Stream (SISD)
- Multiple Instruction Single Data Stream (MISD)
- Single Instruction Multiple Data Stream (SIMD)
- Multiple Instruction Multiple Data Stream (MIMD)

SISD corresponds to the classical von Neumann machine which consists of a storage unit (memory) and a central processing unit (CPU). The CPU executes a single instruction that specifies a sequence of read and write operations on the data stored in memory. SISD does not contain any parallelism. The personal computer (PC) is an example of SISD architecture.

In MISD the processors execute different instructions on the same data. This is not commonly used and applications on this are very few.

Nowadays, most parallel and distributed applications are running on SIMD and MIMD architectures. We briefly describe them below.

2.1.1 Single Instruction Multiple Data

The SIMD model involves executing a single instruction on multiple data sets. A SIMD, as shown in Figure 2.1, consists of n processor elements (PEs) with their own local memories (LM) where it can store data, a global control unit (CU), and an interconnection network. All PEs work under the control of a single instruction stream (IS) issued by CU. There are n data streams (DSs), one per PE. The PEs operate synchronously: at each step, all processors execute the same instruction on a different data in their own

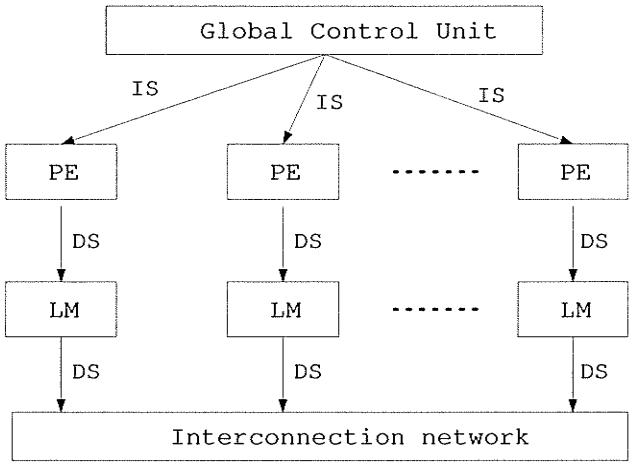

Figure 2.1: The SIMD architecture

local memories.

The first SIMD machine, ILLIAC IV, was developed at the University of Illinois in 1960s. It consisted of a control unit (CU) and 64 PEs. Each PE had two thousand (2K) 64-bit words of memory associated with it (see [7] for detail). The research on ILLIAC IV led ways for constructing more powerful SIMD machines such as the Thinking Machines CM-1 and CM-2.

2.1.2 Multiple Instruction Multiple Data

The class of MIMD computers is the most general and most powerful in Flynn's classification. The MIMD machines, unlike SIMD machines, do not have the global control unit. Each PE has its own control unit and local memory, the PEs communicate with each other via an interconnection network, thereby making them more powerful than those used in SIMD computers (see Figure 2.2). Each PE works under the control of an instruction stream issued by its own control unit. Therefore, when the PEs work together to solve different subproblems of a single problem, they usually operate asynchronously. When all PEs execute the same program on different data, MIMD machines can also be

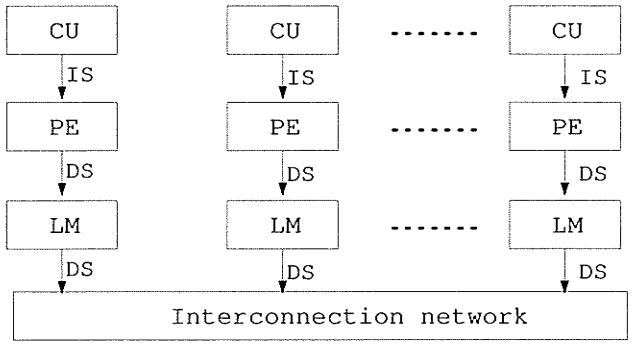

Figure 2.2: The MIMD architecture

referred to as SPMD (Single Program Multiple Data) model. Depending on whether data are communicated implicitly by way of memory storage and retrieval or explicitly from PE to PE, MIMD machines can be categorized as *shared memory machines*(Figure 2.3 (a)) and *distributed memory machines* (Figure 2.3 (b)).

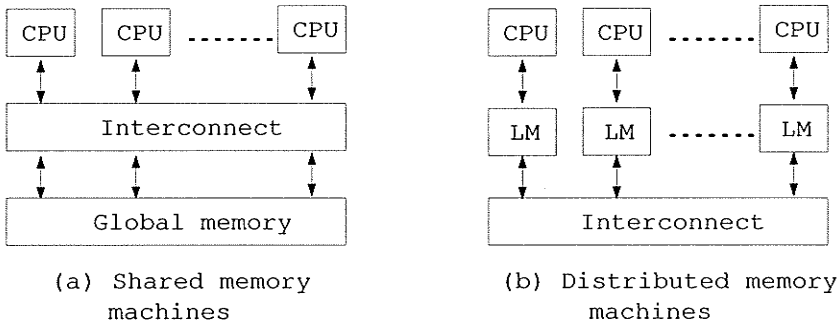

Figure 2.3: Shared memory machines and distributed memory machines

In the shared memory configuration of Figure 2.3(a), the memory is shared in the sense that any of the processors can access the memory locations; that is, the memory is shared by the processors. One drawback of the globally shared memory is that locks and semaphores are necessary to synchronize the access to shared data. Examples of representative machines are SGI Origin 3800 series [62].

In the organization of Figure 2.3(b), processors communicate data to other proces-

sors via message passing and processors can only access their own private memories. If the constitution machines in a distributed system house different hardware and software, the distributed system is known as *heterogeneous*. Distributed memory machines can be composed of common off-the-shelf components and this has been a major contributing factor to their recent popularity. It is possible to extend this design to hold thousands of processors. One drawback of this architecture is its inter-node latency when exchanging data between nodes. Compared with the shared memory machines, the distributed memory machines are inexpensive to build and are very easily scalable.

The class of distributed memory machines is undoubtedly the fastest growing part in the family of high-performance computers nowadays. When designing and implementing algorithms on distributed memory machines, a programmer has to partition and distribute the data over the processors and also the data exchange between processors has to be performed explicitly. Unlike shared-memory systems, the data distribution is completely transparent to the user. However, because the class of distributed memory machines is able to outperform all other types of machines and they are inexpensive to build, this type of machines is very popular in industries and academics. Hence, this is the platform of choice for our current research. As reviewed above, distributed memory environments do not share physical or virtual memory, data are exchanged via message passing. In the follow sections, we review some popular message passing libraries.

2.2 Message Passing Library

In distributed memory machines, every memory module is associated with some individual processor; the processors do not have a common memory. Computing tasks or data are partitioned and distributed explicitly, each running on a separate processor in parallel and communicate with each other through message passing. Message passing may serve different purposes. The most obvious purposes are *communication* and *synchronization* [6]. Communication occurs when a processor requires data from another

processor and must wait for it to arrive. Synchronization arises when processors exchange messages to indicate that they have reached a certain point of program execution or certain requirement has been met.

Message passing is currently the prevailing model in writing performance-oriented applications on a wide variety of distributed memory architectures. When implementing program using message passing model, it is the programmer's responsibility to manage all details of data distribution and task scheduling, load balancing, as well as communication between processors. Since the communication will affect the overall performance, programmers are strongly encouraged to develop algorithms that maximize local computations while minimizing communications. Also, everything is under the programmer's control, the programmer can achieve close to optimum performance if the programmer just spends enough time in performance tuning.

There are two popular message passing libraries that allow programmers to explicitly write message passing programs: Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). We briefly outline them below.

2.2.1 Parallel Virtual Machine

Parallel Virtual Machine (PVM) is a message passing system that enables a network of computers to appear as one large virtual machine to be used as a single distributed memory parallel computer. The PVM project began in early 1990s at Oak Ridge National Laboratory. The overall objective of the PVM system is to enable a collection of computers to be used cooperatively for concurrent or parallel computation. Detailed descriptions and discussions of the concepts, logistics, and methodologies can be found in [28] or online website available at: http://www.csm.ornl.gov/pvm/pvm_home.html.

A portable version of PVM (PVM 2.0) was released in 1991. In 1993, PVM 3.0 was released with a new user application interface (API), which specifically enables PVM applications to run on multiple massive multiprocessors. Meanwhile, PVM research group are trying to keep the PVM interface simple to use and understand. In PVM system,

there is a console through which a user can create and terminate processes at run time on specific hosts and can add and delete hosts. Any process may communicate and/or synchronize with any other.

In PVM system, a computing task is written as a collection of cooperating subtasks. Tasks access PVM resources through a library of standard interface routines. Program is compiled on different hosts and executed concurrently. PVM programs are portable, which implies that you can run a PVM program on a different architecture, once it has been compiled on that system. Besides, a PVM program written in Fortan on one machine can communicate with a C program running on another machine.

PVM is known primarily for its support of multi-platforms, such as UNIX and Windows/ NT. The PVM system has gained widespread acceptance in the high-performance scientific computing community. Despite its popularity, the PVM message passing environment is not a particularly elegant method for expressing many parallel algorithms. PVM has been replaced by MPI in many cases.

2.2.2 Message Passing Interface

Message Passing Interface (MPI) is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users [49]. The official version of the MPI documents can be found at http://www.mpi-forum.org/docs/.

An MPI program consists of autonomous processes, executing their own code in its own address space in an SPMD style. Note that the number of processes can actually be larger than the number of physical processors, since more than one MPI process can be run on each processor. The processes in MPI are ordered and numbered consecutively from 0, the number of each process being known as its rank. The code executed by different processes is identical except for a small number of processes (e.g. the "host" process). All the processes execute their tasks asynchronously. This makes it possible to implement any parallel algorithm. Sometimes, synchronization may be needed between processes depending on the applications under implementation.

All MPI communication calls require a *communicator* argument and MPI processes can only communicate if they share a communicator. Communicator is a key concept used throughout MPI. A communicator consists of a list of processes. The rank of each process identifies each other within the communicator. For example, the rank can be used to specify the source or destination of a message. A default communicator in MPI is MPI_COMM_WORLD which allows all the processes to communicate with each other. In some cases, a programmer can define his own communicators for some special purposes. Hence, a process may belong to several different communicators.

The MPI library contains over 120 routines, making it the richest message-passing library. The routines are used to initialize and terminate the MPI library, to get information about the parallel computing environment, and to send and receive messages. MPI library has been standardized in 1995 as MPI 1.1. Some extensions to MPI 1.1 known as MPI 2.0 were released in 1997. In addition, there exists some freely available, high-quality and portable implementation of MPI such as LAM MPI from Ohio Supercomputing Center and MPICH from Argonne National Laboratory. MPI supports parallel programming in C and Fortran on distributed memory architectures and various platforms such as Unix, Linux, and Windows NT.

2.2.3 Advantages and Disadvantage of MPI and PVM

PVM and MPI are both message passing libraries that can be used for parallel computing. A feature-by-feature comparison of these two libraries is given by Geist et al. [29], and the the reasons why solutions from PVM and MPI are different are stated by Gropp and Lusk [33]. Some relationships between PVM and MPI can be found at [35]. PVM and MPI are message-passing packages providing, in fact, the assembler level of parallel programming for networks of computers [46]. Scientific programmers find that it is tedious and error-prone in writing really complex and useful parallel applications in PVM/MPI [46]. In addition, PVM and MPI are not designed to support development of adaptable parallel applications, that is applications distributing computations and

communications in accordance with the features of underlying heterogeneous machines. These observations by scientific programmers led to the development of mpC.

2.3 mpC

The mpC is a high-level parallel language, which is designed specially to develop portable adaptable application for heterogeneous networks of computers. The main idea underlying mpC is that a programmer can define an abstract network for his/her application and describe in details all the computations and communications to be performed on this abstract network. The mpC programming system uses this information to map the abstract network to any real executing network in such a way that ensures efficient running of the application on this real network. This mapping is performed in run time and based on information about performances of processors and links of the real network, dynamically adapting the program to the executing network.

As a newly invented parallel programming tool, mpC has many advanced features [45]: it allows programmer to define at runtime the total number of participating parallel processes, the total volume of computations to be performed by each of the processes, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. mpC has the following important features [50]:

- Portability: once developed, an mpC application will run as efficiently as possible on any heterogeneous network of computers without any changes of its source code.

- Adaptability: the mpC language allows to write applications adapting not only to nominal performances of processors but also to redistribute computations and communications dependent on dynamic changes of workload of separate computers of the executing network.

- Advancement: mpC is the unique tool having no research or industrial analogs.

(There are some tools executing some functions of an distributed operating system and trying to take into consideration the heterogeneity of processor performances in commodity networks of computers when scheduling tasks in order to maximize throughput of the corresponding network. Unlike such tools, mpC is aimed at minimization of the running time of an application on the executing network. The feature is the most important for end-users, while the network throughput is important for network administrators.)

- Applicability: It is a standard, highly portable and freely available software.

The first version of the mpC programming system for networks of workstations and PCs became available early in 1997. The latest version 2.2.0 was released in November 2001. The current mpC programming environment contains a compiler, run-time support system (RTSS), libraries and a command-line user interface. For detailed information about mpC language, programming environment and samples, please refer to [4, 45, 46] or online website available at http://www.ispras.ru/~mpc/.

The following is an mpC programming example:

```
#include <mpc.h>
#include <stdlib.h>
#define N 3
int [*]main() {
  net SimpletNet (N) mynet;
  char *[mynet]host_name;

  [mynet]host_name=MPC_Get_processor_name();
  [mynet]MPC_Printf(''Hello,world! Host process runs on %s.\n",host_name);
return0;
}
```

The mpC routines are stored in the library mpc.h, and this file must be included in all mpC programs. The number of participating processes, $N = 3$, is defined by the programmer and does not depend on the total number of processes of the parallel program. The

[*] construct before *main* means the main function will be executed by all processes of the parallel program. An abstract network "mynet" is defined which consists of N number of abstract processors. The parallel computations are then described on this network. The execution of this program consists of parallel call to function MPC_Get_processor_name and MPC_Printf by the N processes of the program to which abstract processors of network "mynet" has been mapped. This mapping is performed at runtime.

2.4 Methodology for Creating Parallel Programs

Foster [25] suggests that the methodology of parallel implementations should follow a four step method:

1. Partitioning.
2. Communication.
3. Agglomeration.
4. Mapping.

Partitioning can refer to the decomposition of tasks or the data with which computations are to be performed. The breakdown of the computation into disjoint tasks is termed *functional decomposition*; the partitioning of data amongst the nodes of parallel computation is termed *domain decomposition.* The partitioning stage of a design is intended to expose opportunities for parallel execution.

Communication patterns as noted by Foster [25] are categorized as: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous. In local communication a node communicates within a small set of nodes, while global communication requires that each node have the ability to communicate with all available nodes. Structured communications are built in a regular pattern such as a tree or a grid and do not change over time, while unstructured communications can be an arbitrary arrangement.

Static communication are arranged at initialization, while dynamic communication are arranged at runtime. In synchronous communication, both producers and consumers are aware when communication operations are required, and producers explicitly send data to consumers. In asynchronous situations, nodes need not be synchronized for data transfer.

Agglomeration is the process of combining into coarser-grained tasks, if necessary, to reduce communication requirements or other costs to improve performance. This combining of tasks is also known as increasing the granularity of program structure.

Partitioning, communication and agglomeration lead to the mapping of the parallel program onto a particular architecture. The goal is to maximize local computations while minimizing communications to cut down on total execution time. Foster [25] proposed two conflict strategies to reach this goal: (1) Place tasks that can execute concurrently on different processors, and (2) Place tasks that communicate frequently on same processor. In general, finding optimal solution to this tradeoff is NP-complete, so heuristics are used to find reasonable compromise.

Chapter 3

Monte Carlo and Quasi-Monte Carlo Methods

In the field of computational finance, many problems require numerical evaluation of an integral. However when the dimension of the problem is large, numerical integration methods become intractable. In these cases, the Monte Carlo method is the only practical way to evaluate integrals of arbitrary functions in six or more dimensions [55]. Monte Carlo has been one of the early approaches for option pricing problem [10].

3.1 Monte Carlo Method

In general, Monte Carlo (MC) and quasi-Monte Carlo (QMC) methods are applied to estimate the integral of function $f(x)$ over the $[0,1)^d$ unit hypercube where d is the dimension of the hypercube.

$$I = \int_{[0,1)^d} f(x)dx \tag{3.1}$$

In MC methods, I is estimated by evaluating $f(x)$ at N independent points randomly chosen from a uniform random distribution over $[0,1)^d$ and then evaluating the average

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{3.2}$$

From the law of large numbers, $\hat{I} \rightarrow I$ as $N \rightarrow \infty$. The standard deviation is

$$\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(f(x_i)-\hat{I})^2}. \tag{3.3}$$

The standard error can be estimated as the standard deviation divided by $\sqrt{N}$. Therefore, the error of MC methods is proportional to $\frac{1}{\sqrt{N}}$. In practical implementation, those points are usually generated from a deterministic algorithm. It is expected that these numbers generated in such a deterministic manner to imitate the true randomness of random numbers. These sequences are called *pseudo-random* sequences. Figure 3.1 depicts generation of 500 pairs of random numbers.

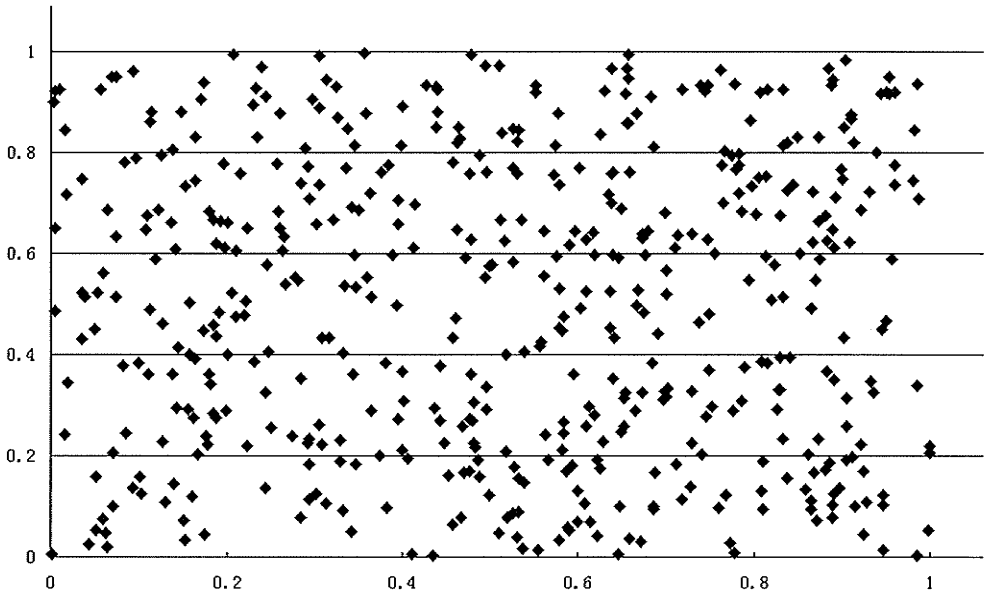

Figure 3.1: Plot of 500 Paris of Random Numbers

It is unnecessary to show random sequences for other dimensions because they will look similar to Figure 3.1, thanks to randomness.

Several advantages make MC method popular among finance and other practitioners. First, MC method is currently the only practical way to deal with numerical integrals

with high dimensionality. Most financial models are high dimensional. For example, pricing a mortgage backed security requires a 360 dimensional space. Current technology only allows the solution of up to 6 dimensions. Second, the standard error of MC simulation does not depend on the dimension of the problems. This property ensures good performance on high-dimensional problems. Last but not least, MC method is easy to apply to many problem and is easy to implement. On the other hand, Monte Carlo method is not perfect, it has several deficiencies that may complicate its usefulness [51]. Limitations include but are not restricted to the following: First, the error of MC methods is proportional to $\frac{1}{\sqrt{N}}$. Hence, decreasing the error by 10 order of magnitude will require an increase in the number of simulation runs N by 100 orders. Second, the MC method is inherently statistical in nature, the result could be wrong, and there are only probabilistic error bounds.

3.2 Quasi-Monte Carlo Method

QMC methods compute the integral (3.1) based on low-discrepancy (LD) sequences. The elements in a LD sequence are "uniformly" chosen from $[0,1)^d$ rather than "randomly". The discrepancy is a measure to evaluate the uniformity of points over $[0,1)^d$. Let $\{q_n\}$ be a sequence in $[0,1)^d$, the discrepancy D_N^* of q_n is defined as follows, using Niederreiter's notation [51]:

$$D_N^*(q_n) = \sup_{B \in [0,1)^d} \mid \frac{A(B, q_n)}{N} - v_d(B) \mid \tag{3.4}$$

where B is a subcube of $[0,1)^d$ containing the origin, $A(B, q_n)$ is the number of points in q_n that fall into B, and $v_d(B)$ is the d-dimensional Lebesgue measure [1]of B. The elements of q_n is said uniformly distributed if its discrepancy $D_N^* \to 0$ as $N \to \infty$. From the theory of uniform distribution sequences [44], the estimate of the integral using a uniformly distributed sequence $\{q_n\}$ is $\hat{I} = \frac{1}{N}\sum_{n=1}^{N} f(q_n)$, as $N \to \infty$ then $\hat{I} \to I$. The

[1]In mathematics, the Lebesgue measure is an extension of the classical notions of length, area or volume to subsets of Euclidean space [67].

integration error bound is given by the Koksman-Hlawka inequality:

$$| I - \frac{1}{N}\sum_{n=1}^{N} f(q_n) | \leq V(f) D_N^*(q_n) \tag{3.5}$$

where $V(f)$ is the variation of the function in the sense of Hardy and Krause [44], which is assumed to be finite. The inequality suggests a smaller error can be obtained by using sequences with smaller discrepancy. The discrepancy of many uniformly distributed sequences satisfies $O((\log N)^d/N)$. These sequences are called low-discrepancy (LD) sequences [51]. Inequality (3.5) shows that the estimates using a LD sequence satisfy the deterministic error bound $O((\log N)^d/N)$. Niederreiter [51] proposed a general principles of generating LD sequences. The best known LD sequences are Halton [34], Sobol [60] and Faure [23]. This is a growing research area and new sequences are being proposed.

The following is an example (see figure 3.2 below) of the first 16 numbers of LD sequence[2] distributed over the interval $[0, 1)$: 0.0000, 0.5000($\frac{8}{16}$), 0.75000($\frac{12}{16}$), 0.2500($\frac{4}{16}$), 0.3750($\frac{6}{16}$), 0.8750($\frac{14}{16}$), 0.6250($\frac{10}{16}$), 0.1250($\frac{2}{16}$), 0.1875($\frac{3}{16}$), 0.6875($\frac{11}{16}$), 0.9375($\frac{15}{16}$), 0.4375($\frac{7}{16}$), 0.3125($\frac{5}{16}$), 0.8125($\frac{13}{16}$), 0.5625($\frac{9}{16}$), 0.0625($\frac{1}{16}$). These values are obtained from the online

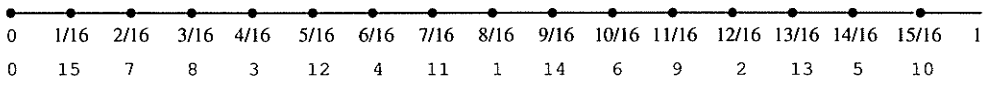

Figure 3.2: The first 16 number of sequence distributed over the interval $[0, 1)$

GNU Scientific Library that generates LD sequence (http://www.gnu.org/software/gsl/). It can be seen that successive points progressively fill-in the spaces between previous points. QMC methods can be viewed as deterministic version of MC methods [51]. The use of LD sequences improves the performance of MC simulations and offers higher accuracy for a similar computational effort compared with standard MC.

LD sequences have been widely used in many disciplines, such as, weather prediction, growth pattern of agriculture. The difference between pseudo-random sequence and

[2]This is a 1-dimensional Sobol sequence. The interval closed at 0 because 0 is included in the sequence, but it is open at 1 because the sequence never reach the number 1.

LD sequences is given by [22]: " Although the ordinary uniform random numbers and quasirandom sequences both produce uniformly distributed sequences, there is a big difference between the two. A uniform random generator on $[0, 1)$ will produce outputs so that each trial has the same probability of generating a point on equal subintervals, for example $[0, 1/2)$ and $[1/2, 1)$. Therefore, it is possible for n trials to coincidentally all lie in the first half of the interval, while the $(n+1)$st point still falls within the other of the two halves with probability $1/2$. This is not the case with the quasirandom sequences, in which the outputs are constrained by a low-discrepancy requirement that has a net effect of points being generated in a highly correlated manner (i.e., the next point "knows" where the previous points are)."

In finance, several examples [1, 11, 27, 53] have shown that the Sobol's sequence is superior to others especially in high dimension problems.

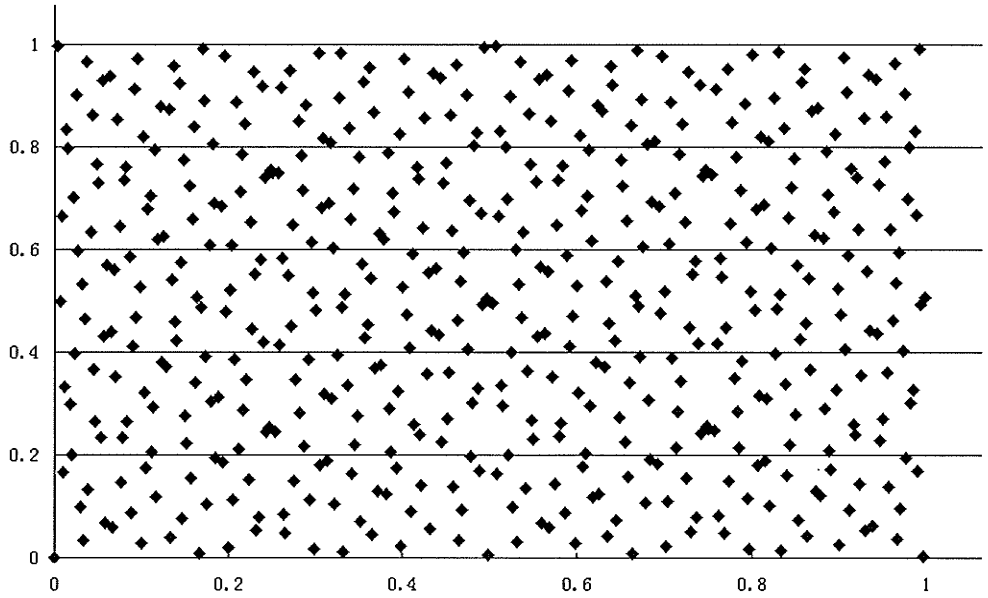

Figure 3.3: Plot of 500 Paris of Sobol LD Numbers

For example, Galanti and Jung [27] observed that "the Sobol sequence outperforms the Faure sequence, and the Faure marginally outperforms the Halton sequence. At 15,000

simulations, the random sequence exhibits an error of 0.07%; the Halton and Faure sequences have errors of 0.1%; and the Sobol sequence has an error of 0.03%. These errors decrease as the number of simulations increases". Hence, in this research, we use Sobol's LD sequence for the QMC simulations. Figure 3.3 plots 500 paris of Sobol LD numbers, compare with Figure 3.1 and notice how the Sobol points are much more evenly distributed but still appear somewhat random.

3.3 Sobol Sequence

Sobol sequence is one of the classical LD-sequences which satisfies smaller discrepancy bounds than others. It has certain advantages over other LD sequences for the computation of high dimensional integrals. This leads to efficient algorithms for pricing option and complex derivative securities.

The Sobol sequences can be viewed as an extension from one-dimensional van der Corput sequence to multi-dimension sequences. The van der Corput sequence is the simplest one-dimensional LD sequence. Let p be any prime number; to obtain the n-th point x_n of the van der Corput sequence, first expand the integer n in terms of p:

$$n = \sum_{i=0}^{m} a_i(n) \times p^i; \tag{3.6}$$

then reflect the expansion in base p about the "decimal point" to get the corresponding quasi-random number:

$$x_n = \phi_p(n) = \sum_{i=0}^{m} \frac{a_i(n)}{p^{i+1}} \tag{3.7}$$

Only a finite number of these $a_i(n)$ will be zero. For example, let $p = 3$ and $n = 11$. We can write 11 in base 3 as:

$$11 = \mathbf{1} \times 3^2 + \mathbf{0} \times 3^1 + \mathbf{2} \times 3^0 \Rightarrow \mathbf{102} \tag{3.8}$$

When reflecting 102 (in base 3) about the "decimal point", we obtain:

$$x_{11} = \phi_3(11) = \frac{2}{3} + \frac{0}{9} + \frac{1}{27} = \frac{19}{27} \tag{3.9}$$

This is clearly a number in $[0, 1)$. The next number in the sequence is $\phi_3(12) = \frac{4}{27}$ and the first 12 number in this sequence, excluding zero, are

$$\{\frac{9}{27}, \frac{18}{27}, \frac{3}{27}, \frac{12}{27}, \frac{21}{27}, \frac{6}{27}, \frac{15}{27}, \frac{24}{27}, \frac{1}{27}, \frac{10}{27}, \frac{19}{27}, \frac{4}{27}\} \tag{3.10}$$

Notice that the new points added tend to fill in the gaps in the existing sequence.

The Sobol sequence use the least prime number 2 as the base. The first dimension of Sobol sequence is a van der Corpt sequence in the base 2 and higher dimensions are permutations of the first dimension following the same procedure. Permutations depend on a set of "direction numbers" v_i which satisfy $v_i = \frac{m_i}{2^i}$ where the m_i are odd positive integers less than 2^i. In order to generate direction numbers, a primitive (irreducible) polynomial over binary arithmetic is selected. This is a polynomial:

$$p(x) = x^q + c_1 x^{q-1} + \cdots + c_{q-1}x + 1, \tag{3.11}$$

with coefficients c_i in $\{0, 1\}$. For dimension j, the v_i^j and m_i^j are generated using the following recurrence formula:

$$m_i^j = 2c_1 m_{i-1}^j \oplus 2^2 c_2 m_{i-2}^j \oplus + \cdots + 2^{q-1} c_{q-1} m_{i-q+1}^j \oplus 2^q m_{i-q}^j, i > d, \tag{3.12}$$

and

$$v_i^j = 2c_1 v_{i-1}^j \oplus c_2 v_{i-2}^j \oplus + \cdots + c_{q-1} v_{i-q+1}^j \oplus v_{i-q}^j \oplus [v_{i-d}^j / 2^d], i > d, \tag{3.13}$$

where $\oplus$ denotes the bit-by-bit exclusive-or operation such that $1 \oplus 0 = 0 \oplus 1 = 1$ and $1 \oplus 1 = 0 \oplus 0 = 0$. The initial value $m_1^j, m_2^j, \cdots, m_q^j$ can be chosen freely provided that each m_i^j is odd and less than 2^j (more details see [31]). The Sobol sequence x_n^j in dimension j is generated by:

$$x_n^j = b_1 v_1^j \oplus b_2 v_2^j \oplus \cdots \oplus b_\omega v_\omega^j \tag{3.14}$$

where ω is number of bits with binary fractions of n. Note that for generating each dimension of Sobol sequence, a different primitive polynomial is needed. This is Sobol's original method. Antonov and Saleev [3] provided a faster method using expression:

$$x_n = g_1 v_1 \oplus b_2 v_2 \oplus g_3 v_3 \oplus \cdots \tag{3.15}$$

where $\cdots g_3 g_2 g_1$ is the binary representation of the Gray code, i. e.,

$$\cdots g_3 g_2 g_1 = \cdots b_3 b_2 b_1 \oplus \cdots b_4 b_3 b_2. \tag{3.16}$$

Formula 3.15 can then be transformed as

$$x_{n+1} = x_n \oplus v_c \tag{3.17}$$

where c is the position of the least significant zero bit in the binary representation of n, i.e., b_c is the least significant zero bit in $\cdots b_3 b_2 b_1$. Bratley and Fox [13] gave an implementation of this method.

3.4 Generating Sobol's LD Sequence

Sobol sequence is one of the classical LD-sequences which satisfies smaller discrepancy bounds than others. Sobol [60] has proposed an algorithm for generating quasi-random sequences. Some discussions about implementing Sobol's algorithm can be found in [13, 40]. Bratley and Fox [13] implemented Sobol's algorithm in Fortran 77. The popular Numerical Recipes (see for example [54]) gives routines implemented in C, Fortran 77, or Fortran 90, but the routines allow the generation of Sobol's sequences in up to six dimensions only. Paskov and Traub [53] from Columbia University implemented a software named FinDer which can generate Sobol sequence up to 370 dimensions, but it is a licensed software.

In this research, we are going to use 10 dimensional (number of time steps, here) Sobol sequence, though we do not have any restriction or limitation in the use of higher dimensions in our algorithm. The whole project will be implemented using mpC which is an extension of ANSI C and the simulations will be performed on machines running Unix/Linux. It will be good for us using C routines which can generate high dimension Sobol sequence. The GNU Scientific Library (http://www.gnu.org/software/gsl/) offers such free routines for generating arbitrary dimensions Sobol sequence. GNU Scientific Library is a collection of numerical routines for scientific computing. The main routines

for generating Sobol sequence are *gsl_qrng_alloc* which returns a pointer to a newly-created instance of Sobol sequence generator and dimension, *gsl_qrng_get* which returns the next point from the sequence generator, and *gsl_qrng_free* which frees all the memory associated with the generator. Using these routines, the LD sequences can be generated in arbitrary dimensions.

Chapter 4

Option pricing and related work

Options on stocks were first traded on Chicago Board of Options Exchange in 1973. Since then, huge volume of options have been widely traded throughout the world. The underlying assets of an option could include stocks, stock indices, foreign currencies, debt instruments and commodities [39]. We introduce in this chapter some basic definitions on option pricing first followed by discussion on some numerical techniques generally used in the literature.

4.1 Definitions

An option is an agreement between two parties to buy or sell an asset at a certain time in the future for a certain price. There are two commonly traded options:

- *Call Option*: A call option [39] is a contract that gives its holder (i.e. buyer) the right to buy a prespecified underlying asset at certain date for a predetermined price without creating an obligation. 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 at any date before its maturity, the option is referred to as an American style Call Option (or American Call).

- *Put Option*: A put option [39] is a contract that gives the right to its holder without creating the obligation, to sell a prespecified underlying asset at certain date for a predetermined price. If the option can be exercised only at its expiration (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 any date before its maturity, the the option is referred to as an American style Put option (or American Put).

The price in the contract is known as *exercise price* or *strike price*; the date in the contract is known as the *expiration date* or *maturity date.*

4.2 Option Pricing

In computational finance, some of the parameters required to price options are: K the strike price; T the life time (expiration date) of the option; S_t the stock price at time t; r the interest rate, μ the drift rate of the stock (a measure of the average rate of growth of the asset price); σ the volatility of the stock; and C the option value.

Here is an example to illustrate the option pricing problem. Suppose an investor enters into a call option contract to buy a stock at price K after six months. After six months, the stock price is S_T. If $S_T > K$ then he can exercise his option by buying the stock at price K, and by immediately selling it in the market he can make a profit of $S_T - K$. On the other hand, if $S_T \leq K$ he is not obligated to exercises the option (that is, buy the stock).

From the above example, an option to buy an underlying asset at time T at price K will get payoff $(S_T - K)^+$, where $(S_T - K)^+ \equiv \max(0, S_T - K)$. Figure 4.1 illustrates the payoff graphically. Similarly, a put option will get payoff $(K - S_T)^+$. Now, the pricing problem is: given the current stock price S, the strike price K, the time to expiry T, risk-free interest rate r, how can one find the present value of the option?

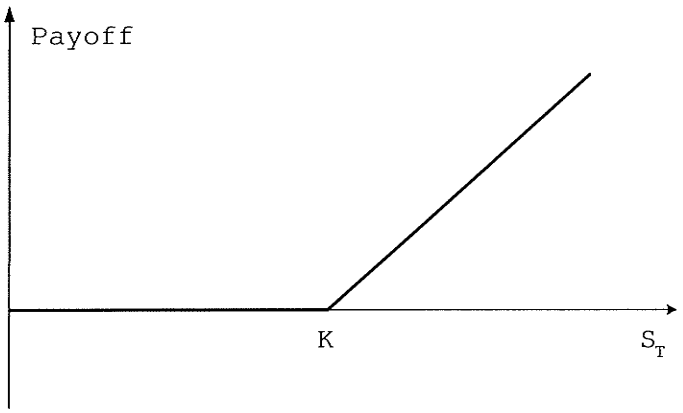

Figure 4.1: Payoff function of the call option

4.3 Related Work

Innovative financial instruments are constantly created in financial markets. Financial models developed to study theses instruments are getting more complex. However, little attention has been paid in the published literature to numerically solve these models to ensure the tractability of these models for various market conditions. Most option pricing models can not be solved to provide exact solution in closed form. Hence, numerical methods have come to play an important role in computational finance. Binomial trees, Finite difference and Monte Carlo simulations are three popular numerical methods that are used to value options.

4.3.1 Binomial Trees

Binomial method is perhaps the simplest and the most intuitive numerical method. It was first proposed by Cox, Ross, and Rubinstein in 1979 [21]. The binomial model for option pricing is based upon a special case in which the price of a stock over some period is assumed to either go up or down by a given proportionate value. When the lattice is built to cover the price movement over the life of the option, it looks like a tree with

root node representing the current date and the leaf nodes representing the possible asset (stock) prices on the maturity date (see Figure 4.2). Each node in the lattice represents a possible price of the underlying asset at a particular point in time.

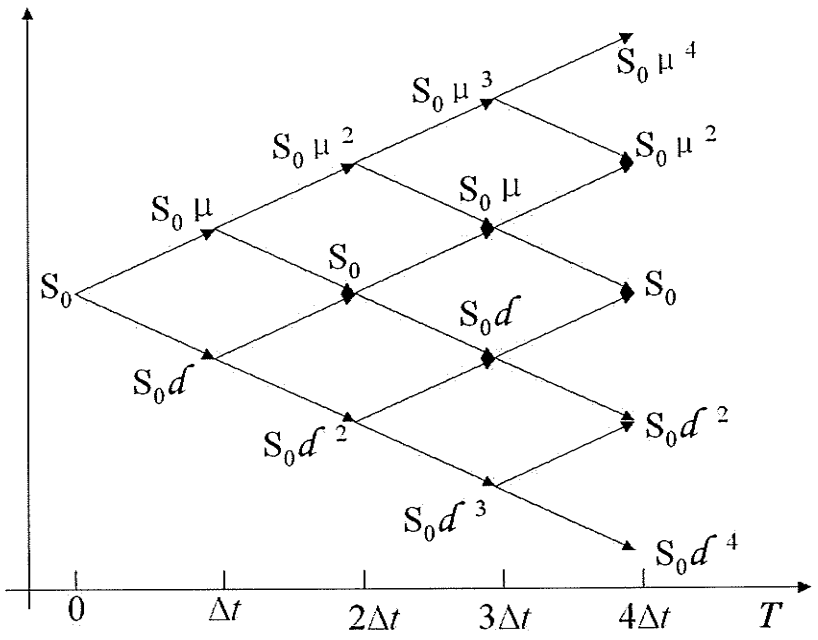

Figure 4.2: A four-step Binomial tree for an asset

Consider a call option on a stock with a current price of S_0 which follows a binomial process. Suppose T is divided into n equal intervals of length $\Delta t = \frac{T}{n}$. At the first time period Δt the asset price can go up to $S_0\,\mu$ with probability p or down to $S_0 d$ with probability $(1-p)$. The parameter μ and d determine the average behavior and the volatility of the asset, ($\mu > 1$; $d < 1$). Cox, Ross, and Rubinstein [21] impose the condition $d = \frac{1}{\mu}$ to force the trees to recombine. Hence, at time $2\Delta t$ for the recombing trees, the possible asset prices are $S_0\,\mu^2$, $S_0\,\mu\,d$, and $S_0 d^2$. The price $S_0\,\mu\,d$ may come from an upward movement followed by a downward movement or from a downward movement followed by an upward movement. Hence, at the i-th ($0 < i \leq n$) time period $i\Delta t$, there

will be $i+1$ possible prices, which we label

$$S_n^i = S_0\,\mu^j\,d^{i-j};\ j = 0, 1, \cdots, i. \tag{4.1}$$

Hence, at the expire time T, there will be $n+1$ possible stock prices. Let C_n^i denote the option value at T corresponding to the $n+1$ stock price, then we have $C_n^i = \max\{S_0\,\mu^j\,d^{i-j} - K, 0\}$. Options are evaluated by starting at time T of the tree and working backward to time zero. Under the risk-neutral measure[1], the value of the j-th node at time $i\Delta t$ before expiry is [39]:

$$C_i^j = e^{-r\Delta t}[pC_{i+1}^{j+1} + (1-p)C_{i+1}^j]. \tag{4.2}$$

Using equation 4.1 and 4.2 we can compute the value of the option at every node at time step $n-1$. We can then reapply equation 4.2 at every node at every time step, working backwards through the tree to compute the value of the option at every node in the tree. This procedure computes the value of the European option at every node in the tree. The parameter μ,d and p are computed using the formula below [39]:

$$\mu = e^{\sigma\sqrt{\Delta t}},\ d = e^{-\sigma\sqrt{\Delta t}},\ p = \frac{e^{r\Delta t} - d}{\mu - d}. \tag{4.3}$$

Binomial model brings useful intuition about complicated problem despite its limitations in terms of accuracy. Various implementations of Binomial method can be found in the literature. For example, Higham [36] summarized nine ways of implementing the binomial method for option valuation in MATLAB. Gerbessiotics [30] introduced an architecture independent parallel binomial tree approach for option price valuations. This algorithm achieves optimal theoretical speedup but it doesn't handle options with multi-assets[2]. Thulasiram and Bondarenko [63] developed and implemented parallel algorithms for pricing options with both single and multi-assets employing binomial lattice approach. Although binomial lattice method is very popular, it has been demonstrated

[1]Risk-neutrality means that the investment on options is assumed to yield at least equivalent to the returns from a bank investment at a fixed interest rate [39].

[2]Multi-assets means an option with several underlying assets

that there are difficulties when this approach is applied directly to many complex option pricing problems (see, for example [12, 37, 64]). In addition, with the increase in the number of state variables, the computational cost of binomial model grows exponentially. The parallel implementation becomes challenging for complex options as explained in [37, 63, 64].

4.3.2 Finite Difference

Finite difference methods are more general than lattice methods and can be applied to price a wide variety of exotic options (see, for example [2, 18, 68, 69]). Finite difference methods value options by solving the differential equations that the options satisfies. The idea is to discretize the differential equation into a set of difference equations and solve the difference equations iteratively [39].

Consider the Black-Scholes partial differential equation for option evaluation [9]:

$$\frac{\partial C}{\partial t} + rS\frac{\partial C}{\partial S} + \frac{1}{2}\sigma^2\frac{\partial^2 C}{\partial S^2} = rC \tag{4.4}$$

To solve this equation, suppose the time T is divided into n equal intervals of length $\Delta t = \frac{T}{n}$. And the stock price S has m steps to reach a maximum value S_{max} which is sufficiently large, then these points define a grid as shown in Figure 4.3. The (i, j) point on the grid is the point that corresponds to time $i\Delta t$ and the stock price $j\Delta S$. We use C_i^j to denote the value of the option at the (i, j) point. In the following, we illustrate an implicit finite difference approach to compute each item of equation 4.4. For an interior point (i, j) on the grid, $\frac{\partial C}{\partial S}$ can be approximated (forward difference) as

$$\frac{\partial C}{\partial S} = \frac{C_i^{j+1} - C_i^j}{\Delta S} \tag{4.5}$$

or backward difference as

$$\frac{\partial C}{\partial S} = \frac{C_i^j - C_i^{j-1}}{\Delta S} \tag{4.6}$$

A more symmetrical approximation is to average equation 4.5 and equation 4.6 which gives (central difference):

$$\frac{\partial C}{\partial S} = \frac{C_i^{j+1} - C_i^{j-1}}{2\Delta S} \tag{4.7}$$

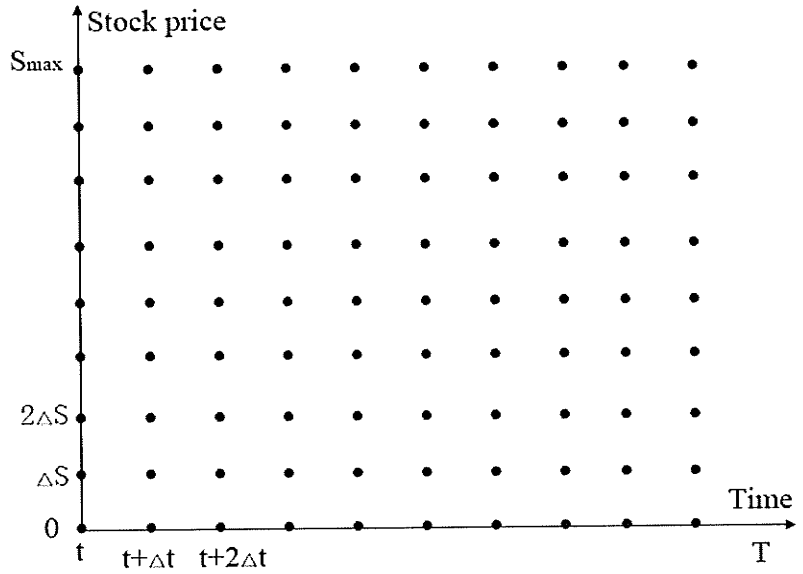

Figure 4.3: Grid for finite difference approach

$\frac{\partial C}{\partial t}$ can be approximated as:

$$\frac{\partial C}{\partial t} = \frac{C_{i+1}^{j} - C_i^j}{\Delta t} \tag{4.8}$$

Similarly, $\frac{\partial^2 C}{\partial S^2}$ at node (i, j) can be approximated as:

$$\frac{\partial^2 C}{\partial S^2} = \frac{\frac{C_i^{j+1} - C_i^j}{\Delta S} - \frac{C_i^j - C_i^{j-1}}{\Delta S}}{\Delta S} \tag{4.9}$$

or

$$\frac{\partial^2 C}{\partial S^2} = \frac{C_i^{j+1} + C_i^{j-1} - 2C_i^j}{\Delta S^2} \tag{4.10}$$

Substituting equations (4.7), (4.8), and (4.10) into the differential equation (4.4) gives:

$$\frac{C_{i+1}^{j} - C_i^j}{\Delta t} + rj\Delta S \frac{C_i^{j+1} - C_i^{j-1}}{2\Delta S} + \frac{1}{2}\sigma^2 j^2 \Delta S^2 \frac{C_i^{j+1} + C_i^{j-1} - 2C_i^j}{\Delta S^2} = rC_i^j \tag{4.11}$$

Hence, for $j = 1, 2, \cdots, m-1$ and $i = 0, 1, \cdots, n-1$, the value at each node of the grid can be computed. Option value can then be obtained by working back from the end of the life of the option to the beginning just like the tree approach.

Finite difference approach was first applied for valuing options in [15, 58]. More recent examples can be found in [18, 19, 65]. There are several finite differencing schemes available in the literature, such as, forward-differencing, backward-differencing, central-differencing, McCormack scheme etc [39]. However, both finite difference methods and binomial lattice methods are computationally intensive and sometimes practically infeasible.

4.3.3 Monte Carlo Simulation

The MC simulation as a numerical method in pricing options was first introduced by Boyle [10] in 1977. Since then, MC simulation has become a popular method for estimating the value of financial options and other derivative securities. There is a vast literature about MC simulation in computational finance. For example, Hull and White [38] employed MC method in stochastic volatility application and obtained more accurate result than using Black-Scholes model [9]; the latter often overprices options about ten percent and the error will be exaggerated as the time to maturity increase. Schwartz and Torous [59] use MC method to simulate the stochastic process of prepayment behavior of mortgage holders and the results matched closely to that actually observed. Fu [26] gives introductory details concerning the use of Monte Carlo simulation techniques for options pricing. Even though the prevailing belief that American-style options cannot be valued efficiently in a simulation model, Tilley [66], Grant et al. [32], as well as Broadie and Glasserman [16] and some others, have proposed MC methods for American-style options and obtained good results. Examples about valuing exotic options can be found in [43]. The literature on MC methods in valuing options keeps growing. More examples can be found in [11, 31, 56]. The traditional MC methods have been shown to be a powerful and flexible tool in computational finance.

4.3.4 Quasi MC Simulation

While the ordinary MC methods are widely applied in option pricing, however, their disadvantages are well-known. In particular, for some complex problems which require a large number of replications to obtain precise results, a traditional MC method using pseudo-random numbers can be quite slow because its convergence rate is only $O(N^{-1/2})$ where N is the number of samples. Different variance reduction techniques have been developed for increasing the efficiency of the traditional MC simulation; such as control variates, antithetic variates, stratified sampling, Latin hypercube sampling, moment matching methods, and importance sampling. For detail about these techniques, please refer to [31]. Another technique for speeding up the MC methods and obtaining more accurate result is to use LD sequences instead of random sequences.

Birge [8] presented how quasi-Monte Carlo sequences can be used in option pricing in 1994 and demonstrated improved estimates through both analytical and empirical evidence. In 1995, Paskov and Traub [53] performed tests about two low-discrepancy algorithms (Sobol and Halton) and two randomized algorithms (classical Monte Carlo and Monte Carlo combined with antithetic variables) on Collateralized Mortgage Obligation (CMO). They obtained more accurate approximations with QMC methods than with traditional MC methods and concluded that for CMO the Sobol sequence is superior to the other algorithms. Acworth et al. [1] compared some traditional MC methods and QMC sequences in option pricing and drew similar conclusion. Boyle et al. [11] also found that QMC outperforms traditional MC and Sobol sequence outperforms other sequences. Galanti and Jung [27] used both pseudo-random sequences and LD sequences (Sobol, Halton and Faure) with MC simulations to value some complex options and demonstrated that LD sequences are a viable alternative to random sequences and the Sobol sequence exhibits better convergence properties than others. Today, QMC methods are successfully used in computational finance as an alternative to MC method. In next chapter, we present details of MC and QMC methods for option pricing followed by parallelization of QMC for option pricing in chapter 6.

Chapter 5

Quasi-Monte Carlo Method for Option Pricing

5.1 Monte Carlo Method

According to Boyle et al. [11], the Monte Carlo methods for option pricing can be divided into three basic steps: (1) Simulate sample paths of the underlying state variables (e.g., underlying asset prices and interest rates) over the relevant time horizon. Simulate these according to the risk-neutral[1] measure; (2) Evaluate the discounted cash flows of a security on each sample path, as determined by the structure of the security in question; (3) Average the discounted cash flows over the sample paths.

The following is a European Call option [39] to be evaluated in this research. This option gives its holder a payoff defined by:

$$\max(0, S_T - K) \tag{5.1}$$

In order to determine the price of the option, a Black-Scholes [9] option pricing model

[1]Risk-neutrality means that the investment on options is assumed to yield at least equivalent to the returns from a bank investment at a fixed interest rate [39].

gives the following stochastic differential equation:

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{5.2}$$

where W is a standard Wiener process (also called Brownian motion). Under the risk-neutral measure, μ is set to $\mu = r$. Equation (5.2) can be rewrite as:

$$\frac{dS_t}{S_t} = rdt + \sigma dW_t, \tag{5.3}$$

This equation may be interpreted as modelling the percentage changes $\frac{dS_t}{S_t}$ in the stock price as the increments of a Brownian motion [31]. The random variable W_t is normally distributed with mean 0 and variance t, it can be simulated by random samples of $\sqrt{t}Z$ where Z is a standard random variable, i.e., $Z \sim (0, 1)$. Knowing the initial value S_0 of the underlying asset, the MC simulation can estimate the value of S_t, and subsequently gives estimation of the payoff from that price.

To simulate the path followed by S, suppose the life of the option has been divided into n short intervals of length Δt ($\Delta t = T/n$), the updating of the stock price at $t + \Delta t$ from t is [39]:

$$S_{t+\Delta t} - S_t = rS_t\Delta t + \sigma S_t Z\sqrt{\Delta t}, \tag{5.4}$$

This enables the value of $S_{\Delta t}$ can be calculated from initial value S_0 at time Δt, the value at time $2\Delta t$ to be calculated from $S_{\Delta t}$, and so on. Hence, a completed path for S has been constructed.

In practice, in order to avoid discretization errors, it is usually to simulate $(\ln S)$ rather than S. From Itô's lemma, the process followed by $\ln S$ of (5.4) is [39]:

$$d\ln S = (r - \frac{\sigma^2}{2})dt + \sigma dz \tag{5.5}$$

so that

$$\ln S_{t+\Delta t} - \ln S_t = (r - \frac{\sigma^2}{2})dt + \sigma Z\sqrt{\Delta t} \tag{5.6}$$

or equivalently:

$$S_{t+\Delta t} = S_t \exp[(r - \sigma^2/2)\Delta t + \sigma\sqrt{\Delta t}Z]. \tag{5.7}$$

Substituting independent samples $Z_1, \ldots, Z_N$ from the normal distribution into (5.7) yields independent samples $S_T^{(i)}$, $i = 1, \ldots, N$, of the stock price at expiry time T. Hence, the option value is given by

$$C = \frac{1}{N} \sum_{i=1}^{N} C_i = \frac{1}{N} \sum_{i=1}^{N} e^{-rT} \max\{S_T^{(i)} - K, 0.0\}. \tag{5.8}$$

The Standard deviation of C is

$$\sqrt{\frac{1}{(N-1)} \sum_{i=1}^{N} (C_i - C)^2}. \tag{5.9}$$

Please note that the several mathematical descriptionis of MC method presented in Section 3.1 is discretely related to this formula. For example, I in equation (3.1) corresponds to C and $f(x)$ in equation (3.1) corresponds to the right hand side of equation (5.8).

The following algorithm illustrates the steps in simulating M paths of n timesteps each.

Algorithm 1 Monte Carlo Algorithm

1. Initialize the parameters such as S, r, K, T, σ,
2. for $i = 1$ to M do /*M = number of simulations. i.e., for each simulation*/
3. for $j = 1$ to n
4. generate standard normal sample
5. simulate sample paths of the stock prices.
6. next j
7. for each simulated path, compute the pay-off of the option.
8. next i
9. Compute the discounted average of above simulated pay-offs.

Table 5.1 gives a schematic illustration of a spreadsheet implementation of this method. The S_{ij} in the spreadsheet denotes the underlying asset price at the j-th timestep along the i-th path. The spreadsheet has M rows where each row is a path of the underlying asset and each path consists n steps. From each path, the spreadsheet computes a value

Path \ Step	1	2	3	...	n		
1	S_{11}	S_{12}	S_{13}	...	S_{1n}		$C_1 = \exp(-rT) * \max(0, S_{1n} - K)$
2	S_{21}	S_{22}	S_{23}	...	S_{2n}		$C_2 = \exp(-rT) * \max(0, S_{2n} - K)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
M	S_{M1}	S_{M2}	S_{M3}	...	S_{Mn}		$C_M = \exp(-rT) * \max(0, S_{Mn} - K)$
							C =Average$(C_1, C_2, \cdots, C_M)$

Table 5.1: A spreadsheet for estimating the expected present value of the payoff of a European call option

of S_{in} and a value of discounted payoff C_i. The C_i are averaged to produce the final result C. In practical implementation, we can view $\exp(-rT)$ as a constant. So in each path, we compute the pay-off of the call option $\max(0, S_{in} - K)$; and finally we compute the discounted average of these simulated pay-offs: $C = \exp(-rT)\text{Average}(C_1, C_2, \cdots, C_M)$.

5.2 A Numerical Example

Suppose there is a one-year maturity European call option with the current asset price at \$20.00 and volatility of 20%. The continuously compounded interest rate is assumed to be 6% per annum, this option pays no dividend. The simulation has 10 timesteps and 100 simulations; $K = 20.00$, $T = 1$, $S_0 = 20.00$, $\sigma = 0.2$, $n = 10$, $M = 100$.

Firstly, the parameters; Δt,$\sigma\sqrt{\Delta t}$ and $\ln S_0$ are pre-computed:

$\Delta t = \frac{T}{n} = \frac{1}{10} = 0.01$;

$\sigma\sqrt{\Delta t} = 0.2 \times \sqrt{0.1} = 0.0632$;

$\ln S_0$ =2.9957;

$(r - \frac{1}{2}\sigma^2)\Delta t = 0.004$.

For each simulation $j = 1$ to M, where $M = 100$, $\ln S_t$ is initialized to $\ln S_0$ =2.9957. Then for each timestep $i = 1$ to 10, $\ln S_t$ is simulated. For example, let ε denote standard

normal sample, for $j = 1$ and $i = 1$:

$\ln S_{11}$

$= \ln S_0 + (r - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t} \times \varepsilon$=2.9957 + 0.004 + 0.0632 * 0.1635 = 3.0101.

and for i=2

$\ln S_{12}$

$= \ln S_{11} + (r - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t} \times \varepsilon$=3.0101 + 0.004 + 0.0632 * 0.0184 = 3.0152.

And so on.

At $i = 10$, $\ln S_{1n} = 3.0338$, $S_T = \exp(\ln S_{1n}) = 20.78$, $C_T = \max(0, S_T - K) = \max(0, 20.78 - 20) = 0.78$

The sum of the values of C_T and the squares of the values of C_T are accumulated: $\Sigma_{j=1}^{M} C_T = 252.59$ and $\Sigma_{j=1}^{M} C_T^2 = 14361.67$. The estimate of the option value is then given by $C = 252.59/100 \times \exp(-0.06 * 1) = 2.3788$. The standard deviation (SD) is: $\sqrt{\frac{1}{(M-1)}\sum_{i=1}^{M}(C_i - C)^2} = 11.08816$ and so the standard error is $\frac{SD}{\sqrt{M}} = 11.08816/10 = 1.108816$. Figure 5.1 illustrated a set of $M = 100$ simulated paths:

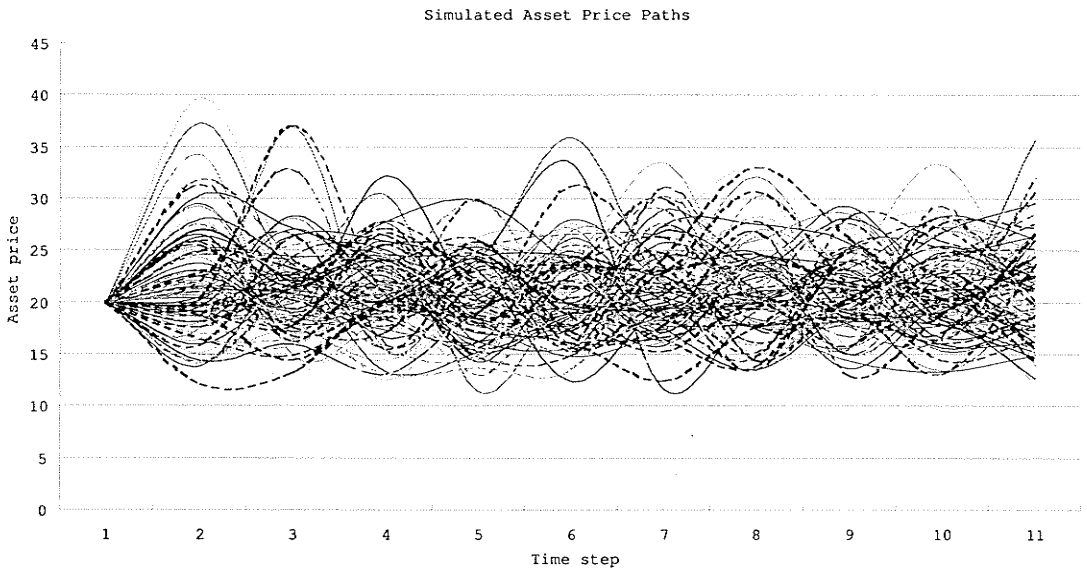

Figure 5.1: Simulated asset price

Table 5.2 illustrates the numerical results of $\ln S_t$ and value of S_t. In the table, SUMCT denotes the sum of the value of the Call option given by each path; SUMCT2 denotes the sum of the squares of the values of the call option given by each path; SD denotes the standard deviation; SE denotes standard error.

In order to get an acceptably accurate estimate of the option price, a very large number of simulations has to be performed, typically in the order of millions [20]. To reduce the computational burden and improve the efficiency of MC simulation, quasi-random numbers are suggested instead of pseudo-random numbers in MC simulation.

5.3 Quasi-Monte Carlo Method

The slow convergence rate, $O(N^{-1/2})$ for N number of samples of the MC method has motivated research in QMC techniques. QMC simulation is the traditional Monte Carlo simulation but using LD sequences. The use of LD sequences in MC simulation improves the performance of MC simulations offering less computational effort or higher accuracy.

Consider generating the LD sequences for the MC simulation of the path of an asset price with n steps as usual. With one source of uncertainty, we can think the number of dimension as the number of discrete time intervals of one sample path. The number of iterations M is the number of sample paths and hence the number of LD points. The pseudo-code is illustrated in Algorithm 2.

Figure 5.2 illustrates the prices obtained from pricing a European call option using pseudo-random numbers and Sobol sequence. It is clear that the prices using the Sobol LD sequences converge much faster as a function of the number of simulations than the prices using pseudo-random numbers.

K	T	S	σ	r	div	N	M				SUMCT	SUMCT2	SD	
20	1	20	0.2	0.0600	0.00	10	100				252.59	14361.67	11.08816	
Δt		$\sigma\sqrt{\Delta t}$	$\ln S$								Call value	SE		
0.1		$0.0632t$	$2.9957S$								2.3788	1.1088		
M / step	0	1	2	3	4	5	6	7	8	9	10	ST	CT	CT*CT
1	2.9957	3.0101	3.0152	3.0730	3.0535	3.0996	2.9495	2.9751	2.9363	2.9342	3.0338	20.78	0.7755	0.60
2	2.9957	3.0881	3.0907	3.1858	3.2242	3.3346	3.2936	3.3046	3.3085	3.3243	3.2594	26.03	6.0331	36.40
3	2.9957	2.9732	2.9209	2.9651	2.9430	2.9232	2.9114	2.8982	2.8676	2.8040	2.6973	14.84	0.0000	0.00
4	2.9957	2.9823	3.0664	3.0776	3.0928	3.0520	3.0670	3.0088	3.0153	3.0369	3.1024	22.25	2.2517	5.07
5	2.9957	2.9316	2.9618	2.9880	3.0156	2.9415	2.9307	2.9508	2.8413	2.7954	2.7989	16.43	0.0000	0.00
6	2.9957	3.0926	3.0709	2.9603	2.9161	2.9837	2.9470	2.9536	3.1133	3.1353	3.1165	22.57	2.5664	6.59
⋮														
96	2.9957	3.0473	3.0546	3.0905	3.1025	3.1132	3.0515	3.1227	3.0699	3.0673	3.0638	21.41	1.4089	1.98
97	2.9957	3.1212	3.1282	3.0204	3.0769	3.0488	2.9282	2.9440	2.9220	2.9152	3.0020	20.13	0.1263	0.02
98	2.9957	3.0711	3.1665	3.0888	3.1363	3.0977	3.1661	3.2475	3.3449	3.2825	3.2784	26.53	6.5344	42.70
99	2.9957	3.0267	3.0844	3.0340	3.1270	3.1296	3.0939	3.1400	3.2443	3.2162	3.1644	23.68	3.6756	13.51
100	2.9957	2.9712	2.9415	2.9759	2.9275	2.8712	2.8355	2.9126	2.8412	2.8686	2.9235	18.61	0.0000	0.00
S	1	2	3	4	5	6	7	8	9	10				
1	21.02	21.08	20.19	24.04	15.68	27.40	20.46	22.71	16.33	26.03				
2	18.60	18.77	17.31	16.18	19.57	17.74	17.66	14.41	13.28	14.84				
3	27.99	17.28	21.15	18.63	17.18	20.83	19.38	22.17	18.88	22.25				
4	14.91	17.48	14.34	19.80	21.04	24.92	23.41	21.24	16.05	16.43				
5	23.73	19.29	24.12	20.85	17.70	24.47	21.22	13.58	20.17	22.57				
⋮														

Table 5.2: Numerical example for Monte Carlo valuation of a European Call option

Algorithm 2 Quasi-Monte Carlo Algorithm

1. Initialize the parameters such as S, r, K, T, σ,
2. generate n dimensional Sobol sequence
3. for $i = 1$ to M do /*M = number of simulations. i.e., for each simulation*/
4. for $j = 1$ to n
5. simulate sample paths of the stock prices using Sobol sequence.
6. next j
7. for each simulated path, compute the pay-off of the option.
8. next i
9. Compute the discounted average of above simulated pay-offs.

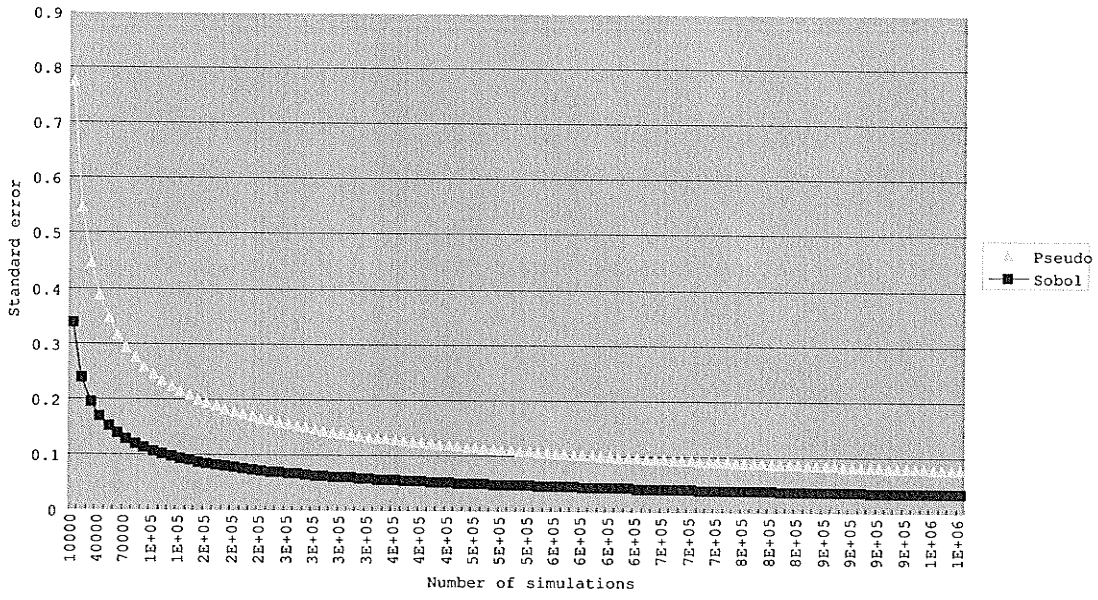

Figure 5.2: Relative pricing error for a European Call option using pseudo-random numbers and Sobol LD numbers

Chapter 6

Parallel Quasi-Monte Carlo Method for Option Pricing

Monte Carlo techniques lend itself easily to parallelization since each simulation can be implemented independent of each other. Most serial MC codes are readily adaptable to a parallel environment as explored in one of the application of MC [56]. One approach is to have each processor execute the sequential algorithm with different pseudo random number sequences and combine the final results from all the processors at the end of the execution. Other techniques involve parallelizing the pseudo random number generators [55, 61].

QMC technique converges faster than MC method and has proven to be advantageous in a number of financial applications (see for example [1, 8, 11, 27, 48, 53]). However, the issues of parallelizing QMC are different from MC parallelization due to the deterministic nature of LD sequences. There has been some recent interests in parallelizing QMC [17, 48, 52, 57]. Given P processors, if we select N_i points on processor P_i ($0 \leq i \leq P-1$), then the discrepancy of these points must be similar to the discrepancy of $\Sigma_{i=0}^{P-1} N_i$ points [61]. We need to have the results of parallel computation same as the results of the sequential computations. There are three popular techniques for parallelizing LD sequences [61]: *Leapfrog*, *Blocking*, and *Parameterization*.

Leapfrog is a method to assign N elements of the sequence to P processors in the same way as a deck of cards is dealt in turn to players in a card game. Let $x_0, x_1, \cdots, x_N$ be the elements of the sequence. Using this method, the processor i will have elements $x_i, x_{i+P}, x_{i+2P}, \cdots$ where $0 \leq i \leq P-1$. The disadvantage of this method is that even if the elements of the original sequence have low correlation, the elements of the leapfrog subsequence may be correlated. Also, it has been noted [61] that in the *Leapfrog* method, the processors have to be synchronized after a few iterations to get accurate results. This adds to the computational cost.

Blocking is a scheme with which the elements of a sequence are divided into equal-sized blocks [61]. Suppose there are P processors and $x_1, \cdots, x_N$ be an m-dimensional random sequence (each element x_i is m-dimensional vector). With *blocking* scheme, each processor will be assigned N/P elements. The elements in each block are contiguous, each block has size $B = N/P$. Therefore, the processor i will have the elements $x_{iB}, x_{iB+1}, x_{iB+2}, \cdots$ where $0 \leq i \leq P-1$. This scheme also suffers from the drawback of required synchronization [61].

Parameterization of LD sequence is in some sense similar to that of parameterization in parallelizing random number sequence, where independent sequences are used on each processor. In general, it is difficult to avoid inter-processor correlations [5].

All above strategies try to obtain maximum speedup by evenly distributing computations over available processors. On a homogeneous network, where all machines have the same processing speed, distributing N/P points is reasonable. However, in practice, the above strategies will result in poor performance even among homogeneous networks. This is because, processors might run at different speeds; some may also be used for other computations and may be involved in other communications. Hence, those faster processors or light-load processors will finish their computation tasks quickly and wait for slower ones at points of synchronization; the overall computation time will be determined by the time elapsed on the slowest processor. Because of synchronization of processors, the most powerful processors will run at the speed of the slowest processor.

It is not a desirable scenario since the parallel computing does not take full advantage of the potential computing power.

Srinivasan [61] compares the effectiveness of these three strategies in pricing financial derivatives and concluded that *blocking* is the most promising method if there are a large number of processors running at unequal speeds. However, the disadvantages of *blocking* scheme are well pronounced. First, if a processor consumes more LD elements than it was assigned, then the subsequences could overlap. Second, if a processor consumes less LD elements than it was assigned, then some elements will be wasted. The final result will be the same as the sequential computation that use the same LD sequence with some "gaps". Hence, a good parallel MC algorithm should distribute computations based on the actual performance of processors at the moment of the execution of the program. The more powerful a processor, the more tasks it will be assigned and will be able to handle. That is, data, computations, and communications should be distributed unevenly among processors to achieve the best execution performance. In the following sections, we discuss development of such an algorithm.

6.1 Tasks Partition

An ideal parallel QMC algorithm should distribute computation tasks to processors proportional to processors' actual computing powers. Otherwise, the load of processors will be unbalanced, resulting in poor performance.

Having known the power of each processors, the computation tasks assigned to each processor can be computed by performing the following partition algorithm (Algorithm 3). With this partition algorithm, the QMC simulations are unevenly distributed across processors based on the actual performance of each processor.

Having known the tasks of each processor, we can decide the points will be executed by each processor. Suppose each simulation consumes q elements of the given LD sequence, and processor i has t_i tasks, then the whole number of elements will be consumed

Algorithm 3 Partition

1. Given N is the total tasks, p is the number of processors, $power_i$ is the i-th processor's power. then, the tasks assigned to the i-th processor is:

$$task_i = [N \times \frac{power_i}{\sum_{i=0}^{p} power_i}] \tag{6.1}$$

2. After step1, if there are tasks left, then assign them to host processor.

by processor i is $B_i = t_i \times q$. Note that B_i is not necessarily N/P where N is the number of points and P is the number of processors. Hence, the LD sequence is partitioned into uneven blocks. This partition of LD sequence is somewhat like the general *blocking* scheme, but it is superior to general *blocking* scheme, in which the LD sequence is partitioned into equal size or the burden is on the programmer to determine the block size B. In the literature, B is usually chosen to be greater than N/P to avoid overlapping of subsequences. Therefore, there is a chance that the result produced from the *blocking* scheme of QMC is not the same as that of a sequential computing [61]. Using Algorithm 3, there is no overlapping in sub-sequences and that the sequential run and the parallel run results match.

To implement this algorithm, the program must provide information of the entire computing space and relative performances of actual processors in the run time. Currently, no parallel programming tool can implement such parallel algorithm except mpC. As a new parallel programming tool, mpC offers a very convenient way to obtain the statistics information of the computing space and the power of each processor. By using mpC, a programmer can explicitly specify the uneven distribution of computations across parallel processors.

6.2 mpC

The mpC programming tool is specially designed for writing high performance parallel computation programs on networks of heterogeneous computers; it is an extension of

the ANSI C language. In section 2.3 we introduced some features of mpC. For detailed information about mpC language, programming environment and samples, please refer to [4, 45, 46] or online website available at http://www.ispras.ru/~mpc/.

The mpC offers mechanisms that other parallel programming languages do not have, through which a programmer can describe (dynamically) a virtual network topology for the application under study. At run time, the mpC environment will map the virtual network to real executing network based on information about performances of processors and links of the real network. This can be done by defining a *network object* in mpC program. A *network object*, or simply *network* is a basic notation of mpC language, which comprises virtual processors and links. The *network* in mpC program is just like a user-defined datatype in general programming language. Allocating network objects and discarding them is performed in similar way as allocating data objects and discarding them. In mpC, a processor which creates the *network* is called a *parent* of the created network. For example, the type declaration

```
/*line 1 */    nettype Ring(n) {
/*line 2 */      coord I = n;
/*line 3 */      link {
/*line 4 */            I > 0: [I] ⟷ [I − 1];
/*line 5 */            I == 0:[0] ⟷ [n − 1];
/*line 6 */        };
/*line 7 */      parent [0];
/*line 8 */     };
```

introduces a topology named *Ring* that corresponds to networks consisting of n processors interconnected with undirected links in a ring structure. Note that the real network is not necessarily connected as a ring. The *Ring* is the topology of the application under study. The Line 1 is a header of the network type declaration. It introduces the name of the network type. Line 2 is a coordinate declaration, declaring the coordinate system to which processors are related. It introduces the integer coordinate variable I ranging from 0 to $n-1$. Line 3 to line 6 are a link declaration. They specify links between

processors. Line 4 stands for the predicate: for all $I > 0$ there exists undirected links connecting processors with coordinates $[I-1]$ and $[I]$, and line 5 stands for the predict: for $I == 0$ there exists an undirected link connecting processors with coordinates [0] and $[n-1]$. Line 7 is a parent declaration. It specifies that the parent has coordinate [0].

With the network type declaration, one can declare a network object identifier of this type. For example, the declaration:

net Ring(5) r

introduces the identifier r of the network object of the type Ring. Once the network is defined, a programmer can start describing parallel computation on the network. Section 6.3 gives detail information on implementing the parallel QMC algorithm using mpC.

6.3 Implementation Detail

Suppose we have m processors doing QMC simulations. One processor (host processor) distributes tasks to the other processors and collects their results. In QMC simulations, each simulation is independent, there is no communications between processors except with the host processor. Hence, this QMC simulations has a *star* topology where the host processor is the central node and the other processors (nodes) connected directly to the central node. The computing and communication are based on this topology. The following *network* declaration describes this topology.

```
/*line 1 */    nettype Star(m, p[m]) {
/*line 2 */      coord I = m;
/*line 3 */      node {
/*line 4 */        I ≥ 0: p[I];
/*line 5 */        };
/*line 6 */      link {
/*line 7 */        I > 0: [0] ↔ [I − 1];
/*line 8 */        };
```

```
/*line 9 */        parent [0];
/*line 10 */       };
```

The header (Line 1) introduces parameters of the topology *Star*, namely, the integer parameter m and the vector parameter p consisting of m integers. Vector p is used to store the relative performances of the m processors. Line 2 introduces a coordinate declaration declaring the coordinate system to which virtual processors are related. The integer coordinate variable I ranges from 0 to $m-1$. Lines 3–5 are node declaration. Line 4 stands for the predicate (for all $I \geq 0$), that the virtual processor, whose relative performance is specified by the value of $p[I]$, is related to the point with coordinate $[I]$, and so on. Lines 6–8 are link declaration, which specify links between virtual processors. Line 7 stands for the predicate for $I > 0$ and $I < m$ there exists undirected links connecting virtual processors with coordinates [0] and [I-1]. Line 9 is a parent declaration. It specifies that the parent has coordinate [0]. After the *network* is created in mpC program,

Algorithm 4 Parallel Quasi-Monte Carlo Algorithm

1. Initialize the parameters such as S, r, K, T, σ,
2. Compute relative performances of actual processors
3. Partition tasks according to the performance of each processor (Algorithm 3)
4. Assign elements of LD sequence to processors according to their tasks (Scatter blocks)
5. Broadcast options' parameters
6. Execute the sequential algorithm on each processor
7. Gather the results of each processors
8. Produce the final result on host processor.

it executes the rest of computations and communications. A call to library function *MPC_Processors_static_info* made on the entire computing space returns the number of actual processors and their relative performances. Based on relative performances of actual processors, Algorithm 3 computes the number of simulations which should be computed by every processor. Then the subsequences for each processor can be determined using the method mentioned in section 6.1. Further, the steps to follow

are: **a)** broadcast option's arguments; **b)** scatter subsequences, **c)** perform sequential computing on each processor, and finally **d)** the host processor gather the results from each processor and produce the final result. Algorithm 4 illustrates this procedure.

The following functions are part of the mpC code implementing the QMC simulations:

```
/* 1 */ void [*]Simulation(float *[host]option, float *[host]sequence,
/* 2 */                     float *[host]results, int [host]n)
/* 3 */ {
/* 4 */       repl nprocs, tasks[MAXNPROCS], dn;
/* 5 */       repl double *powers;
/* 6 */       dn=n;
/* 7 */   MPC_Processors_static_info(&nprocs, &powers);
/* 8 */   Partition(nprocs, powers, tasks, dn);
/* 9 */   {
/* 10*/     int [host]i;
/* 11*/     for(i=0; i<[host]nprocs; i++)
/* 12*/       ([host]printf)("proc=%d___tasks=%d\n", i, ([host]tasks)[i]);
/* 13*/   }
/* 14*/   {
/* 15*/     net Star(nprocs, tasks) mynet;
/* 16*/     float *[mynet]doption, *[mynet]dsequence, *[mynet]dresults;
/* 17*/     repl [mynet]n;
/* 18*/     int [mynet]myn, [mynet]sof;
/* 19*/
/* 20*/     sof=[mynet](sizeof(float));
/* 21*/     n=[mynet]dn;
/* 22*/     myn=([mynet]tasks)[I coordof doption];
/* 23*/
/* 24*/     dsequence=([mynet]calloc)(myn*steps, sof);
/* 25*/     ([host]free)([host]dsequence);
/* 26*/     [host]dsequence=(void *)sequence;
/* 27*/
/* 28*/     doption=([mynet]calloc)(6, sof);
/* 29*/     ([host]free)([host]doption);
```

```
/* 30*/      [host]doption=(void *)option;
/* 31*/
/* 32*/      dresults=([mynet]calloc)(1, sof);
/* 33*/      ([host]free)([host]dresults);
/* 34*/      [host]dresults=(void *)results;
/* 35*/
/* 36*/      ([([mynet]nprocs)mynet])
/* 37*/      ParCompute(doption, dsequence, dresults, [mynet]tasks);
/* 38*/    }
/* 39*/ }

/* 40*/ void [net SimpleNet(p)v]
/* 41*/       ParCompute(float *doption, float *dsequence,
/* 42*/       float *dresults, repl *r)
/* 43*/ {
/* 44*/    repl s=0;
/* 45*/    int myn, i;
/* 46*/    int *d, *nold, c,*disp,*rct;
/* 47*/
/* 48*/    myn=r[I coordof r];
/* 49*/    ([(0)v])MPC_Bcast(&s, doption, 1, 6, doption, 1);
/* 50*/
/* 51*/    d=calloc(p, sizeof(int));
/* 52*/    nold=calloc(p, sizeof(int));
/* 53*/    disp=calloc(p, sizeof(int));
/* 54*/    rct=calloc(p, sizeof(int));
/* 55*/    for(i=0, d[0]=0; i<p; i++)
/* 56*/      {
/* 57*/        nold[i]=r[i]*steps;
/* 58*/        disp[i]=i;
/* 59*/        rct[i]=1;
/* 60*/        if(i+1<p)
/* 61*/          d[i+1]=nold[i]+d[i];
/* 62*/      }
```

```
/* 63*/     c=nold[I coordof c];
/* 64*/    ([(0)v])MPC_Scatter(&s, dsequence, d, nold, c, dsequence);
/* 65*/    ([v]SeqCompute)(doption, dsequence, dresults, myn);
/* 66*/    ([(0)v])MPC_Gather(&s, dresults, disp, rct, 1, dresults);
/* 67*/ }
```

Line 1 defines function *Simulation* with four arguments belonging to the virtual host processor: pointer **option** to the the option arguments, pointer **sequence** to the Sobol LD numbers, pointer **results** to the computing results from parallel processors, and **n** the simulation numbers. Function *Simulation* is called a basic function. In mpC, there are three types of functions: *basic*, *network*, and *nodal* functions. *Basic function* can be called and executed on the entire computing space. Only in basic functions networks can be defined. *Network function* is called and executed on a network object. *Nodal function* can be called and executed by any virtual processor. In mpC, the ANSI C functions are considered nodal functions.

Line 4 defines integer variable **nprocs** and array **tasks** and integer **dn**. The three variables are declared *replicated* (repl) over the entire computing space. In mpC, the keyword *repl* means all distributions of the value of the variable equal to each other. Line 5 define pointer **powers** distributed over the entire computing space and specifies that it points to a replicated data object. Line 7 calls library nodal function *MPC_Processors_static_info* on the entire computing space returning the number of actual processors and there relative performances. After this call the variable **nprocs** will hold the number of actual processors, and replicated array **powers** will hold the relative performances. Line 8 calls function *Partition* which computes the number of tasks of each processor based on their performances. Line 10 to line 12 prints the partition results.

Line 15 defines a network object **mynet** which is an instance of the network *Star* and executes most of the rest of computations and communications. It consists of **nprocs** virtual processors, the computing task of the i-th virtual processor being characterized by the value of **tasks[i]**. Having known the computing tasks of each processor, we can compute the number of LD numbers needed for each processor. Variable **myn** denotes the

number of simulation assigned to each processor. After execution line 22, each component of **myn** will contain the number of simulations of corresponding virtual processor. Line 24 allocates memory for variable **dsequence** which will be used to store the assigned LD numbers of each processor. Line 28 allocates memory for variable **doption** which is used to store the Option's arguments. Line 32 allocates memory for variable **dresults** which is used to store the computing results of the virtual processor. Then at line 36, call the parallel computing function *ParCompute*.

Function *ParCompute* is a basic function, which has four arguments belonging to the virtual processor: pointer **doption** to the Option's arguments, pointer **dsequence** to the LD numbers, pointer **dresults** to the computing result, and replicated variable pointer r to the tasks. In lines 40-42, the header of the definition of function *ParCompute* declares identifier v of a network being a special network formal parameter of the function. In the function body, special formal parameter p is treated as an unmodified variable of type int replicated over network v. p holds the number of virtual processor of network v. The rest of formal parameters of the function are distributed over v. Line 48 gives the computing tasks of a virtual processor with coordinate i.

Line 49 calls to the embedded network function MPC_Bcast which is declared in the header as follows:

```
int [net SimpleNet(n) w]MPC_Bcast(
    repl const *source,
    <s_type> *s_buffer,
    int const s_step,
    repl const count,
    <d_type> *d_buffer,
    int const d_step);
```

This call broadcasts option's arguments from the parent of v to all virtual processors of v. As a result, each component of the distributed array pointed by **doption** will contain this option's arguments.

Statements in lines 51-64 are asynchronous. They form four p-member arrays **d**,

nold, **disp**, and **rct** distributed over v. After this, **nold**[i] will hold the number of LD numbers assigned by Partition algorithm for virtual processor with coordinate i, and **d**[i] will hold the displacement which corresponds to this portion of LD numbers. **disp**[i] holds the displacement which corresponds to the computing result from virtual processor i. **rct**[i] holds the receive counts from virtual processor i. Line 63 is also asynchronous. After this execution, each component of c will hold the number of LD numbers which will be used by corresponding virtual processor.

Line 64 calls to network function MPC_Scatter which is declared as:

```
int [net SimpleNet(n) w]MPC_Scatter(
      repl const *source,
      <s_type> *s_buffer,
      int const *disps,
      int const *lengths,
      repl const count,
      <d_type> *d_buffer);
```

This call scatter LD sequence from the parent of v to all virtual processors of v. As a result, each component of **doption** will point to an array containing the corresponding portion of LD sequence. Line 65 is to execute a sequential computing, and finally line 66 is to gather the computing results by calling network function MPC_Gather which is declared as follows:

```
int [net SimpleNet(n) w]MPC_Gather(
      repl const *destination,
      <d_type>  *d_buffer,
      int const *disps,
      int const *lengths,
      repl const count,
      <s_type> *s_buffer);
```

This call gathers results from each virtual processor of v.

We have presented the most interesting part of the mpC code implementing the parallel QMC algorithm for option price. We will present the experimental results and comparison with other implementations in next chapter.

Chapter 7

Results and Evaluation

In this chapter, we present the analytical results first followed by the experimental results.

7.1 Analytical Results

We first show that the performance of our parallel QMC algorithm is better than general parallel algorithms (e.g. *blocking* scheme).

Suppose we have m processors doing N parallel QMC simulations. Let us denote the m processors as $p_1, p_2, \cdots, p_m$. Without loss of generality, we sort the processors in ascending order based on their processing speed as follows:

$$p_1 \leq p_2 \leq p_3 \cdots \leq p_m. \tag{7.1}$$

p_1 is the slowest processor, and p_m is the fastest. We denote the processing power of processor i as $power_i$. Then the tasks t_i assigned to processor i using formula 6.1 is $t_i = [N \times \frac{power_i}{\sum_{i=0}^{m} power_i}]$, hence, we must have

$$t_1 \leq t_2 \leq t_3 \cdots \leq t_m. \tag{7.2}$$

Because the tasks assigned to each processor is proportional to its speed, the load of each processor is balanced. This means that the processors will finish their tasks at the same

amount of time. Let us denote the computing time by T_{cp}, and communication time T_{cm}. Then the overall computing time is

$$T_{total} = T_{cp} + T_{cm} \tag{7.3}$$

Now let us consider the general parallel schemes doing parallel QMC on the same architecture. In general schemes, each processor will be assigned the same number of tasks N/m. Due to varying processing speeds of the processors, the time spent on each processor is different. The overall computing time is determined by the slowest processor. In the case of 7.1, p_1 is the slowest processor. Suppose p_1 spends T'_{cp} time to finish its task. We assume the communication time is still T_{cm}. This assumption is acceptable for our parallel QMC algorithm, due to (1) in parallel QMC simulations, the communication operations' contribution to the total execution time of the algorithm is negligibly small compared to that of the computation; and (2) in parallel QMC simulations, parallel processes do not communicate frequently sending and receiving messages. Hence, the overall computing time using general schemes is

$$T'_{total} = T'_{cp} + T_{cm} \tag{7.4}$$

To show $T'_{total} \geq T_{total}$, we need to show $T'_{cp} \geq T_{cp}$. We know T'_{cp} and T_{cp} are the times the processor p_1 takes to execute the tasks. Hence, to show $T'_{cp} \geq T_{cp}$, we only need to show $N/m \geq N \times \frac{power_1}{\sum_{i=0}^{m} power_i}$, that is to show $\frac{1}{m} \geq \frac{power_1}{\sum_{i=0}^{m} power_i}$.
Suppose $\frac{power_1}{\sum_{i=0}^{m} power_i} > \frac{1}{m}$, then we must have:

$$\begin{aligned}
N &= t_1 + t_2 + \cdots + t_m \\
&= N \times \tfrac{power_1}{\sum_{i=0}^{m} power_i} + N \times \tfrac{power_2}{\sum_{i=0}^{m} power_i} + \cdots + N \times \tfrac{power_m}{\sum_{i=0}^{m} power_i} \\
&= N \times (\tfrac{power_1}{\sum_{i=0}^{m} power_i} + \tfrac{power_2}{\sum_{i=0}^{m} power_i} + \cdots + \tfrac{power_m}{\sum_{i=0}^{m} power_i}) \\
&> N \times (\tfrac{1}{m} + \tfrac{1}{m} + \cdots + \tfrac{1}{m}) \\
&= N \times (m \times \tfrac{1}{m}) = N.
\end{aligned}$$

We get contradiction $N > N$, hence $\frac{power_1}{\sum_{i=0}^{m} power_i} > \frac{1}{m}$ is impossible. We must have $\frac{1}{m} \geq \frac{power_1}{\sum_{i=0}^{m} power_i}$. Then $N \times \frac{1}{m} \geq N \times \frac{power_1}{\sum_{i=0}^{m} power_i}$, so $T'_{cp} \geq T_{cp}$. That is, the general

parallel schemes will spend more time than our parallel algorithm running on the same architecture.

7.2 Experimental Results

This section presents some results of experiments of the QMC algorithm presented in Chapter 6. In our experiments, we price a one-year maturity ($T = 1$) at the money European call option with current asset price, $S = \$20$, and strike price $K = \$20$; the risk-free rate of interest is 6 percent ($r = 0.06$), and the volatility of the asset is set at twenty percent ($\sigma = 0.2$) and the asset pays no dividend. We divide the time period over which we wish to simulate S_T into 10 intervals (timesteps=10). We save these parameters into a file. For each experiment including MPI implementations, we use the same input parameters.

We decide to run 1,000,000 simulations. For each simulation, we need a 10 dimensional Sobol point. Hence, we should have 1,000,000 Sobol points, each point is 10-dimension. Since these Sobol points are deterministic, it is unnecessary to generate them each time for every experiment. We save the 1,000,000 points into a file. The file size is about 81 mega bytes! Saving option's parameters and LD points into files makes our programs more flexible: we can value different options without changing mpC code; and we can use different LD sequences and even random sequence.

A small local network of 7 Solaris workstations (named Cadmium01, Cadmium02, Cadmium03, Cadmium04, Cadmium05, Cadmium06, and Cadmium08) and 18 Linux machines (canary-01, $\cdots$, canary-08, and 10 other bird-named machines) running Fedora Core 2 are used for the experiments. Here we present the experiment results on the 7 Solaris workstations [1]. The names of the workstations are manually written in a VPM (virtual parallel machine) file. A VPM file contains the real distributed memory machines.

[1]There exists cross-platform problems between Solaris and Linux machines in mpC version 2.3.0 [42]. The mpc runs fine on individual clusters.

The format of a vpm file is:

machine_name number_of_processes.

The contents of our vpm file are:

cadmium01 1

cadmium02 1

⋮

cadmium08 1

We set the number of processes as 1 to each workstation, because we will compare the performance of our algorithm with those implemented using MPI. In MPI, though more than one MPI process can be run on each processor, it is difficult for a programmer to assign a fixed number of processes running on a specified workstation.

By executing the mpC command *mpccreate vpmfile*, we obtain the initial static structure information of the network, which is saved in the form of ASCII file as the following:

```
# cadmium01
s1 p883 n1 c1 c1 c1
# cadmium02
s1 p930 n1 c1 c1 c1
# cadmium03
s1 p879 n1 c1 c1 c1
# cadmium04
s1 p999 n1 c1 c1 c1
# cadmium05
s1 p959 n1 c1 c1 c1
# cadmium06
s1 p870 n1 c1 c1 c1
# cadmium08
s1 p899 n1 c1 c1 c1
```

This information will be used by mpC runtime system (RTS). Here, each computer is

Processors	p1	p2	p3	p4	p5	p6	p7
Performance	883	930	879	999	959	870	899

Table 7.1: Relative performance of 7 heterogeneous workstations

characterized by six parameters. The first parameter, s, shows the number of processors. $s1$ means the computer has only one processor. Thus, in our experiments, all the workstations are uniprocessor computers.

The second parameter, p, shows the performance of the computer. We can see that *cadmium04* is the most powerful computer. Note that at each time when one executes the command *mpccreate vpmfile*, the performance values are different; and at runtime the execution of the **recon** statement updates the value of the parameter for each participated computer. Hence, we should know that each time the same program running on these computers will spend different time.

The third parameter, n, shows the total number of processes of the parallel program to run on the computer. In our experiments, each computer runs 1 process.

Finally, the last three parameters are used by mpC system to determine the speed of point-to-point data transfer between processes running on the same computer as a function of size of the transferred data block [45]. The first of them specifies the speed of transfer of a data block of 64 bytes, and the second and the third specify that of 64^2 and 64^3 bytes respectively. The speed of transfer of an arbitrary size data block is calculated by interpolation of the measured speeds. In our experiments, each computer has only one process, so there is no communication between processes running on the same computer.

Using above parameters, we measure the time to compute an option price. Timing is obtained via the mpC wall clock function, *MPC_Wtime()*. We don't include the time the program takes in reading the LD sequence file and arguments file. For convenience, we list the relative performance of the 7 processors in Table 7.1. The 1,000,000 simulations are distributed to the 7 processors based on their performance as shown in Table 7.2

Processors	p1	p2	p3	p4	p5	p6	p7
# of Tasks	137563	144882	136937	155631	149400	135535	140052

Table 7.2: Distribution of 1,000,000 simulations to 7 heterogeneous workstations

	number of processors						
simulations	1	2	3	4	5	6	7
100000	0.02	0.13	0.24	0.35	0.44	0.60	0.43
200000	2.24	2.06	1.95	1.76	1.32	1.04	0.76
300000	5.26	3.18	2.05	1.94	1.87	1.40	1.21
400000	7.11	5.41	3.25	2.44	2.10	1.78	1.34
500000	9.48	6.71	4.30	2.71	2.34	2.19	1.68
600000	11.89	8.10	5.92	3.84	2.64	2.45	1.89
700000	14.22	8.12	6.51	4.11	2.99	2.91	2.28
800000	17.25	9.60	7.29	4.97	3.07	3.11	3.07
900000	19.11	10.25	8.02	5.62	3.31	3.29	3.16
1000000	20.13	11.60	8.49	5.71	4.82	3.53	3.31

Table 7.3: Time to do QMC simulations in seconds

using Algorithm 6.1. The computing time is 3.3054 seconds. For comparison purpose, we have implemented a sequential QMC algorithm and obtained the running time with the same input parameters and LD sequence.

Table 7.3 lists the the computing time of different combinations of simulations and processors. Note that for different number of simulations, we have different LD sequence files. The experiments are started from single workstation *Cadmium01*. We add other computers (*Cadmium02*, *Cadmium03*, $\cdots$, and *Cadmium08*) to increase the applications performance. In the table, the single processor is the machine *Cadmium01*; the 2 processors are *Cadmium01* and *Cadmium02*; 3 processors are *Cadmium01*, *Cadmium02*, and

Cadmium03; and so on. A more intuitive figure is given in Figure 7.1.

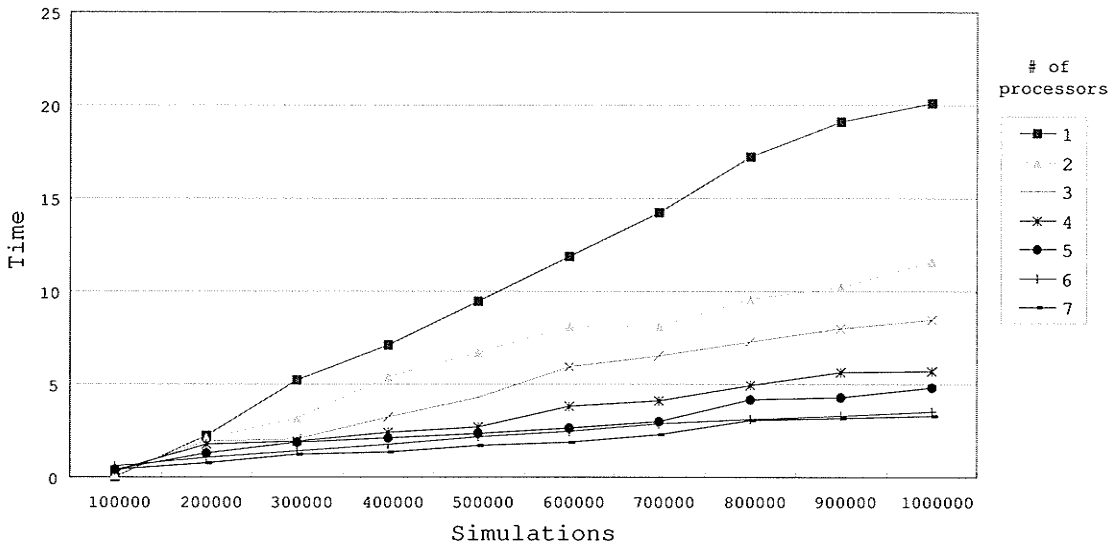

Figure 7.1: Execution time with respect to various processors and simulations

From Figure 7.1 and Table 7.3, we notice that, initially, with a 100,000 simulations, the single processor machine gives better results than seven processors. However, for large input sizes, having more number of processors is beneficial. With 1,000,000 simulations, seven processors gives an execution time of 3.31 seconds compared to 11.60 seconds with two processors. This is a significant decrease in execution time.

Figure 7.2 illustrates the speedup achieved by using different processors. Note that the running time of the mpC program substantially depends on the workload of the workstations. We can see that the speedup curves have some ups and downs. This is due to the fact that the workload of each workstation is dynamic (the workstations are accessible to other students in Computer Science department). At certain time, the workstations are involved in other computations, and the low performance of some workstations substantially increases the whole computing time. We took the average

of the execution times after running the simulations for several iterations. With seven processors, the relative speedup is approximately 6x. Overall, the mpC program ensures good speedup.

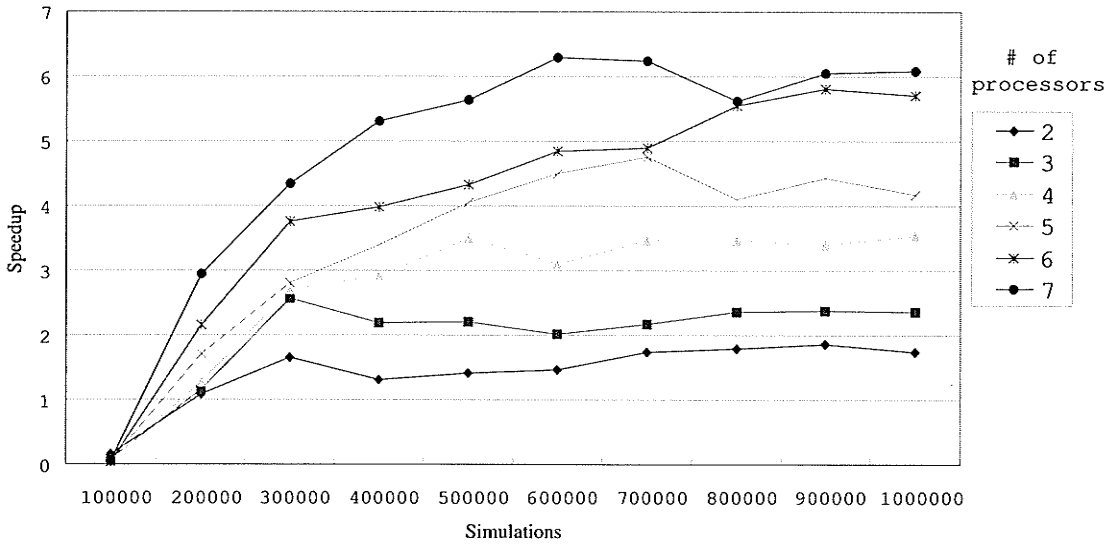

Figure 7.2: Speedups computed relative to sequential code running on workstation Cadmium01

To get a better estimation of our mpC program, we developed two versions of the MPI programs: 1) static distribution tasks among processors (general blocking (BK) scheme); 2) a manager-workers (MW) approach which simulates load balancing scheme to some extent.

Using general blocking (BK) scheme, the tasks (N) and the LD sequence are equally distributed among the m processors. In this experiment, usually the number of processors must be a factor of the number of simulations (i.e. N/m is an integer); otherwise, the result will be different from a sequential algorithm's result using the same arguments. In section 7.1, we analytically showed that this scheme is not efficient on heterogeneous

environment where the processors exhibit different processing speeds and resources comparing to our parallel QMC algorithm. Algorithm 5 and Algorithm 6 represents the manager-worker's algorithm.

Algorithm 5 Manager

1. Initialize parameters: Options arguments, LD sequence
2. Broadcast Option's arguments
3. While work> 0 do
4. Receive a result from any worker and dispatch a new worktag together with a new subsequence;
5. Manager assign tasks to itself and consume necessary subsequences;
6. Record each processor's tasks.
6. Next
7. Receive results for outstanding work requests.
8. Tell all the workers to exit.
9. Print result.

Algorithm 6 Worker

1. Do
2. Accepts work requests
3. Do sequential computing
4. Returns results
5. Until a termination request is received.

Table 7.4 gives the execution time for 1,000,000 simulations on 7 processor on three different implementations: mpC, blocking scheme (BK) with MPI and manager-worker scheme (MW) with MPI. The table also indicates the number of tasks distributed to each processor in each of the three schemes. In BK scheme, since the task distribution is static, each processor receives 142857 tasks. In the MW scheme, each processor is assigned one task per request. It's interesting to see in the table that though processor 4

Processors' ID	Processor's performance	mpC	MPI	
			BK	MW
1	883	137563	142857	500000
2	930	144882	142857	133597
3	879	136937	142857	81659
4	999	155631	142857	77377
5	959	149400	142857	74096
6	870	135535	142857	67771
7	899	140052	142857	65500
time (Sec.)		3.31	4.67	95.48

Table 7.4: Time to do 1,000,000 simulations using three schemes

has better computing performance than processor 1, processor 1 is assigned more number of tasks in total. This is done by the scheduler. The MW scheme in MPI does not take the performance of the processors into consideration. Finally, with mpC, we notice that the processors are given tasks according to their performance. Processor 4 gets the most number of tasks, 155631, since it is the fastest; while processor 6 gets 135535 number of tasks since it is the slowest. From experiments, we find the mpC scheme outperforms BK scheme because the load of mpC scheme is balanced; the BK scheme outperforms MW scheme because there are too many communications in MW scheme, though MW scheme simulates load balance to sone extent. Overall, our algorithm implemented in mpC is the most efficient one while the MW scheme is the most inefficient one since the workload is unbalanced and communications dominate the whole computing (sending requests and results to manager, receiving tasks from manager).

In the MW program, instead of assigning one task per request, we tried to assign arbitrary number of tasks to each request, thereby simulating mpC to some extent. Let pt be the number of tasks assigned to a processor per request. When $pt = 1$, the

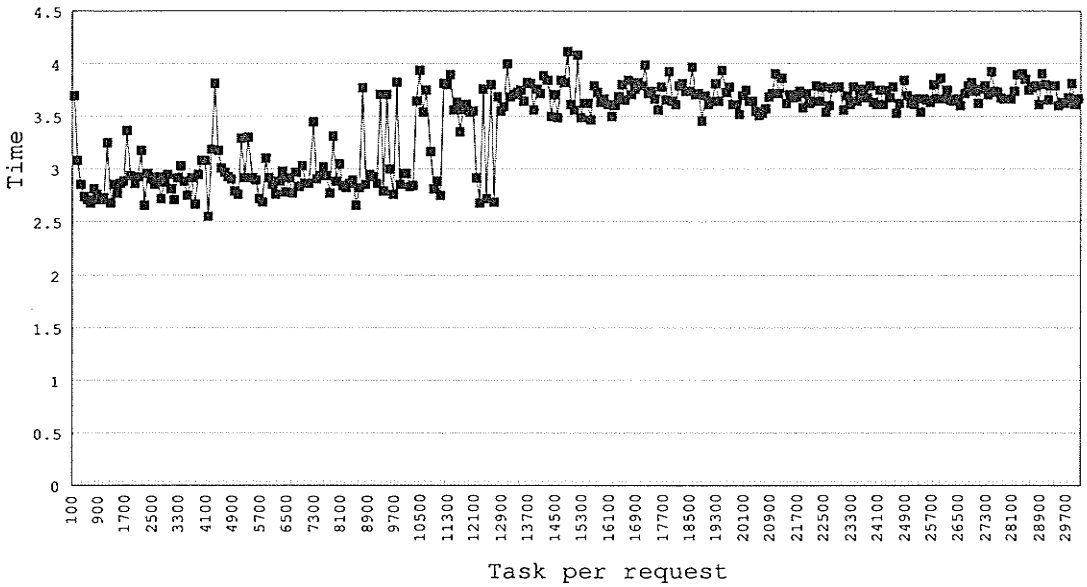

Figure 7.3: Manager-worker scheme with different amount tasks per request

computing time is 95.48 seconds as seen in Table 7.4. We experimented from $pt = 100$ till $pt = 30000$, with 100 tasks in each increment. Figure 7.3 illustrates the test results.

From the test, we found in some cases the MW performs better than mpC program and in some cases it does not. For example, when $pt = 500$, the runtime is 2.70764 seconds; and when $pt = 4300$, the run time is 3.81360 seconds. We cannot find any trend about the value of pt that will give the best performance. Hence, when designing MW algorithm, it would be tedious to find a suitable value for pt which is user-defined. In addition, the MW scheme is not portable when some conditions are changed. For example, if the number of processors is changed or the workload of some processors is changed, the pt value must also be changed manually. Unlike mpC program, the number of tasks will be automatically changed based on the processors' performance when the number of processors is changed.

From the experiments, advantages of using mpC programming system are especially

clear when programming for heterogeneous distributed memory machines. The mpC parallel language allows the programmer to define all the main features of the algorithm under study, such as, the total number of participating parallel processes, the total volume of data to be transferred on each process, and the topology of the application. In addition, mpC system has its own mapping algorithms to ensure each process to perform computations at the speed proportional to the volume of computation it performs. Hence, these features lead to a more balanced and faster parallel program.

Chapter 8

Conclusions and future work

In this research, we presented a distributed parallel QMC algorithm for pricing options that is adaptable to heterogenous network of workstations. The parallel algorithm distributes the data depending on the architectural features of the machines. We used the Sobol LD sequence for the QMC technique and implemented the algorithm in mpC. Its good performance is justified theoretically and verified experimentally.

As discussed in chapter 6, QMC simulations are well suited for parallel computing. Simulations can be performed on different processors, and the results can be combined finally. Traditional parallel methods try to obtain maximum speedup by evenly distributing computations over available processors, where the underlying features of the machines are not considered while scheduling the tasks. On heterogeneous networks, performance gains are potentially available for algorithms if they are designed to fully exploit the hardware features. This is the very peculiarity of our parallel algorithm, which takes into account the actual performances of both processors and communication links. By comparing with other parallel algorithms and implementations, the speedups exhibited by our algorithm presented in this thesis are promising.

The implementation has demonstrated that mpC and its programming environment are suitable tools for implementing adaptive algorithms on heterogeneous networks. Like MPI, mpC supports writing efficient parallel program running on specified distributed

memory machines. Like MPI, the mpC program is portable to other distributed memory machines without rewriting and recompiling. Unlike MPI, the mpC and its programming environment can specify application performance model and ensure the efficiency while porting program to other distributed memory machines. An outstanding feature of using mpC is that a programmer can specify the topology of the application under study and mpC system can map the topology to real network system based on processors' processing speeds and network bandwidths in run time. Some other features of mpC programming system are that such as it allows both data parallelism and task parallelism as well as vector computing. All these factors make mpC programming system unparalleled among known parallel programming tools.

In future, we would like to extend our parallel algorithm to price American and Asian-style options and options with multi-assets. For American-style options, one has to deal with the possibility of an early exercise to achieve an optimal value. For Asian-style options, one has to calculate the average value of the underlier on a specific set of dates during the life of the option. For option with multi-asserts, one has to take into account the correlation between securities.

In this research, we use Sobol sequence in our QMC method. However, other LD sequences may have advantages over Sobol sequences. In particular, Joy, Boyle and Tan [41] use Faure sequences to value a range of complex derivative securities and obtain good results. The application of LD sequences to problems in finance is also a topic of current interest [41]. Future work will consider using alternative LD sequences for option pricing and comparison of their performance and accuracy.

Although this research has concentrated on QMC method for option pricing problem, the technique used in this research can be adopted to other areas of QMC research. We believe our research will offer new understanding of QMC methods and would open up a new venue for QMC application developers in other areas of research.

Bibliography

[1] P. Acworth, M. Broadie, and P. Glasserman. A comparison of some Monte Carlo and quasi Monte Carlo methods for option pricing. In H. Niederreiter, P. Hellekalek, G. Larcher, and P. Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1996*, pages 1–18. Springer-Verlag, Berlin, 1998.

[2] J. Andreasen. The pricing of discretely sampled Asian and lookback options: a change of numeraire approach. *The Journal of Computational Finance*, 2(1):5–23, 1998.

[3] I. A. Antonov and V. M. Saleev. An economic method of computing $LP_{T-sequences}$. *USSR Computt. Math. Phy.*, 19:252–256, 1979.

[4] D. Arapov, V. Ivannikov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis. A parallel language for modular distributed programming. In *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis*, pages 248–255, Aizu, Japan, March 1997. IEEE Computer Society.

[5] Online article. Parallel Random Number Generators. *Scientific Computing at NPACI*, 3(7), March 1999. Available at http://www.npaci.edu/online/v3.7/SCAN1.html.

[6] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In R. Dierstein, H. Wacker, and D. Mller-Wichards, editors, *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 61–88, Bonn, Germany, 1988. Springer-Verlag New York, Inc.

[7] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, 8(17):746–757, August 1968.

[8] J. R. Birge. Quasi-Monte Carlo approaches to option pricing. Technical report 94–19, Department of Industrial and Operations Engineering,University of Michigan, 1994.

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

[10] P. Boyle. Options: A Monte Carlo approach. *Journal of Financial Economics*, 4:323–338, 1977.

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

[12] P. P. Boyle and S. H. Lau. Bumping up against the barrier with the binomial method. *Journal of Derivatives*, 1:6–14, 1994.

[13] P. Bratley and B. L. Fox. Algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14(1):88–100, 1988.

[14] M. J. Brennan and E. E. Schwartz. The valuation of American put options. *Journal of Finance*, 32:449–462, 1977.

[15] M. J. Brennan and E. S. Schwartz. Finite Difference Methods and Jump Processes Arising in the Pricing of Contingenet Claims: A Synthesis. *Journal of Financial and Quantitative Analysis*, pages 461–474, September 1978.

[16] M. Broadie and P. Glasserman. Pricing American-Style Securities Using Simulation. Technical Report 96–12, Columbia - Graduate School of Business, 1996. available at http://ideas.repec.org/p/fth/colubu/96-12.html.

[17] B. C. Bromley. Quasirandom Number Generators for Parallel Monte Carlo Algorithms. *Journal of Parallel and Distributed Computing*, 38(0132):101–104, 1996.

[18] A. Chhabra, P. Thulasiraman, and R. K. Thulasiram. FLEET: A Framework for evaLuating European options in parallEl and disTributed environment. *Intl.J. of High Performance Computing and Applications*, (to appear).

[19] I. J. Clark. Option pricing algorithms for the Cray T3D supercomputer. In *Proceedings of Computational and Quantitative Finance '98 Conference*, New York, September 1998.

[20] L. Clewlow and C. Strickland. *Implementing Derivatives Models.* John Wiley & Sons, New York, 1998.

[21] J. Cox, S. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Financial Economics*, 7:229–263, 1979.

[22] E. W. Weisstein et al. Quasirandom Sequence. From MathWorld–A Wolfram Web Resource. Available at http://mathworld.wolfram.com/QuasirandomSequence.html.

[23] H. Faure. Discrepance de suites associees a un systeme de numeration (en dimension s). *Acta Arithmetica*, 41:337–351, 1982.

[24] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, C-21(9):948–960, September 1972.

[25] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, USA, 1st edition, 1995.

[26] M. C. Fu. Pricing of financial derivatives via simulation. In C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman, editors, *In Proceedings of the 1995 Winter Wimulation conference*, pages 126–132, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, 1995.

[27] S. Galanti and A. Jung. Low-discrepancy sequences: Monte Carlo simulation of option prices. *Journal of Derivatives*, 5(1):63–83, 1997.

[28] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[29] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[30] A. V. Gerbessiotis. Architecture independent parallel binomial tree option price valuations. *Parallel Computing*, 30:301–316, 2004.

[31] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer, New York, 2004.

[32] D. Grant, G. Vora, and D. Weeks. Path-Dependent Options: Extending the Monte Carlo Simulation Approach. *Management Science*, 43(11):1589–1602, Nov 1997.

[33] W. D. Gropp and E. Lusk. Why are PVM and MPI so different? In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, 1997.

[34] J. H. Halton. On the efficiency of certain quasirandom sequences of points in evaluating multidimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[35] R. Hempel. The status of the MPI messsage-passing standard its relation to PVM. volume 1156 of *Lecture Notes in Computer Science*, pages 14–21. Springer Verlag, 1996.

[36] D. J. Higham. Nine ways to implement the binomial method for option valuation in MATLAB. *SIAM Review*, 44(4):661–677, 2002.

[37] K. Huang and R. K. Thulasiram. Parallel algorithm for pricing American Asian options with multi-dimensional assets. In *Proc. 19th Intl. Symp. High Performance Computing Systems and Applications (HPCS)*, pages 177–185, Guelph, ON, Canada, May 2005.

[38] J. Hull and A. White. The Pricing of Options on Assets with Stochastic Volatilities. *Journal of Finance*, 42:281–300, 1987.

[39] J. C. Hull. *Options, Futures, and Other Derivatives.* Prentice Hall, Upper Saddle River, NJ, 5th edition, 2003.

[40] Stephen Joe and Frances Y. Kuo. Remark on algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 29(1):49–57, 2003.

[41] C. Joy, P. P. Boyle, and K. S. Tan. Quasi-Monte Carlo Methods in Numerical Finance. *Management Science*, 42(6):926–938, June 1996.

[42] A. Kalinov. mpc problem with cross-platforming. Personal email communications.

[43] A. G. Z. Kemna and A. C. F. Vorst. A Pricing Method for Options Based on Average Asset Values. *Journal of Banking and Finance*, 14:113–129, 1990.

[44] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences.* John Wiley & Sons, New York, 1974.

[45] A. Lastovetsky. Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Computing*, 28:1369–1407, 2002.

[46] A. Lastovetsky. *Parallel Computing on Heterogeneous Networks.* John Wiley & Sons, 2003.

[47] C. Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches.* John Wiley & Sons, Inc., New York, USA, 2001.

[48] J. X. Li and G. L. Mullen. Parallel computing of a quasi-monte carlo algorithm for valuing derivatives. *Parallel Computing*, 26(5):641–653, 2000.

[49] ANL Mathematics and Computer Science. The Message Passing Interface (MPI) standard. 2005. http://www-unix.mcs.anl.gov/mpi/.

[50] mpC Team. About mpC Parallel Programming Environment. Online article. Available at http://www.ispras.ru/~mpc/mpc_descr.html.

[51] H. Niederreiter. *Random Number Generation and Quasi–Monte Carlo Methods*, volume 63 of *CBMS-NSF Regional Conference Series in Appl. Math.* SIAM, Philadelphia, PA, 1992.

[52] G. Okten and A. Srinivasan. Parallel quasi-Monte Carlo methods on a heterogeneous cluster. In H. Niederreiter et al., editor, *Proceedings of Fourth International Conference on Monte Carlo and Quasi-Monte Carlo*, pages 406–421, Hong Kong, 2000.

[53] S. H. Paskov and J. F. Traub. Faster valuation of financial derivatives. *Journal of Portfolio Management*, 22(1):113–120, Fall 1995.

[54] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Combridge University Press, New York, second edition, 1992.

[55] M. J. Quinn. *Parallel programming in C with MPI and OpenMP.* McGraw-Hill, 2004.

[56] S. Rakhmayil, I. Shiller, and R. K. Thulasiram. Cost of Option Pricing Errors Associated with Incorrect Estimates of the Underlying Assets Volatility: Parallel Monte Carlo Simulation. *IMACS J. Mathematics and Computers in Simulation*, (under review).

[57] W. Schmid and A. Uhl. Techniques of parallel quasi-Monte Carlo integration with digital sequences and associated problems. *Mathematics and computers in simulation*, 55:249–257, 2000.

[58] E. S. Schwartz. The Valuation of Warrants: Implementing a New Approach. *Journal of Financial Economics*, 4(1):77–93, 1977.

[59] E. S. Schwartz and W. N. Torous. Prepayment and the Valuation of Mortgage-Backed Securities. *Journal of Finance*, 44(2):375–392, 1989.

[60] I. M. Sobol. On the distribution of points in a cube and the approximate evaluation of integers. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[61] A. Srinivasan. Parallel and distributed computing issues in pricing financial derivatives through quasi monte carlo. In *Proc. (CD-RoM) 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, FL, USA, 2002. IEEE Computer Society.

[62] SGI. Technical Publications. *SGI - C Language Reference Manual*, June 2003.

[63] R. K. Thulasiram and D. A. 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*, pages 306–313, Vancouver, BC, Canada, August 2002.

[64] R. K. Thulasiram, L. Litov, H. Nojumi, C. Downing, and G. R. Gao. Multithreaded Algorithms for Pricing a Class of Complex Options. In *Proceedings (CD-RoM) of the IEEE/ACM International Parallel and Distribued Processing Symposium (IPDPS)*, San Francisco, CA, USA, April 2001.

[65] R. K Thulasiram, C. Zhen, and A. Gumel. A second order L_0 stable algorithm for evaluating European options. *Intl. J. of High Performance Computing and Networking (IJHPCN)*, (to appear).

[66] J. A. Tilley. Valuing American Options in a Path Simulation Model. *Transactions of the Society of Actuaries*, 45:83–104, 1993.

[67] E. W. Weisstein. Lebesgue Measure. From MathWorld–A Wolfram Web Resource. Available at http://mathworld.wolfram.com/LebesgueMeasure.html.

[68] R. Zvan, P. A. Forsyth, and K. R. Vetzal. Robust numerical methods for PDE models of Asian options. *Journal of Computational Finance*, 1(2):39–78, 1998.

[69] R. Zvan, K. R. Vetzal, and P. A. Forsyth. PDE methods for pricing barrier options. *Journal of Economic Dynamics and Control*, 24(11-12):1563–1590, 2000.