A Parallel Particle Swarm Optimization Algorithm for Option Pricing

A thesis presented

by

Hari Prasain

to

The Faculty of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Science

Computer Science

The University of Manitoba

Winnipeg, Manitoba

Thesis advisor

Dr. Parimala Thulasiraman and

Dr. Ruppa K. Thulasiram

Author

Hari Prasain

A Parallel Particle Swarm Optimization Algorithm for Option Pricing

Abstract

Financial derivatives play significant role in an investor's success. Financial option is one form of derivatives. Option pricing is one of the challenging and fundamental problems of computational finance. Due to highly volatile and dynamic market conditions, there are no closed form solutions available except for simple styles of options such as, European options. Due to the complex nature of the governing mathematics, several numerical approaches have been proposed in the past to price American style and other complex options approximately.

Bio-inspired and nature-inspired algorithms have been considered for solving large, dynamic and complex scientific and engineering problems. These algorithms are inspired by techniques developed by the insect societies for their own survival. Nature-inspired algorithms, in particular, have gained prominence in real world optimization problems such as in mobile ad hoc networks. The option pricing problem fits very well into this category of problems due to the ad hoc nature of the market. Particle swarm optimization (PSO) is one of the novel global search algorithms based on a class of nature-inspired techniques known as swarm intelligence.

In this research, we have designed a sequential PSO based option pricing algorithm using basic principles of PSO. The algorithm is applicable for both European and American options, and handles both constant and variable volatility. We show that our results for European options compare well with Black-Scholes-Merton formula. Since it is very important and critical to lock-in profit making opportunities in the real market, we have also designed and developed parallel algorithm to expedite the computing process.

We evaluate the performance of our algorithm on a cluster of multicore machines that supports three different architectures: shared memory, distributed memory, and a hybrid architectures. We conclude that for a shared memory architecture or a hybrid architecture, one-to-one mapping of particles to processors is recommended for performance speedup. We get a speedup of 20 on a cluster of four nodes with 8 dual-core processors per node.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I want to thank my advisors, Dr. Parimala Thulasiraman and Dr. Ruppa Thulasiram, for their guidance during my research and study at the University of Manitoba. My very sincere thanks to them for the support, encouragement, motivation, and inspiration that they provide me during the course of this thesis. I also thank them for every other great effort they put into training me in the scientific field. Without their support and motivation, it is impossible to think of this output.

I am also pleased to thank the Advisory committee members (Dr. Pourang Irani and Dr. Srimantoorao S. Appadoo) for their valuable positive suggestions and advices.

I express my deepest gratitude to my beloved grandmother "Mrs. Subhadra Devi Prasain" and my parents, "Mr. Bharat Raj Prasain" and "Mrs. Chayya Devi Prasain", for their never-ending support, endless encouragement, and unconditional love. I also, especially, thank my uncle "Mr. Laxmi Prasad Prasain", aunt "Anita Prasain", sister "Kalpana Prasain" and brother-in-law "Bhupendra Singh" for all the advices, motivation and support they gave me, when I needed them the most.

I also extend thanks to all of my friends at University of Manitoba. I want to thank them for all their help, support, and interest and friendly gestures. In particular, I thank "Dr. Girish K. Jha", "Mr. Moazzam Khan", "Mr. Sameer Kumar", "Mr. David Allenetor", and "Sean Gustafson" for their valuable suggestions and advices provided to me along the way.

I sincerely thank the Faculty of Science, Faculty of Graduate Studies and the department of computer science of the University of Manitoba for providing scholarships and assistantship during my study.

Last but not least I thank all my family members and the other people who have

supported me along the way.

This thesis is dedicated in the memory of my late grandfather Bed Prasad Prasain.

Chapter 1

Introduction

Option pricing is one of the computationally challenging problems in finance. A *call* option is a contract that gives the right to its *holder* (i.e. buyer) without any obligation to buy a pre-specified underlying asset at a pre-determined contract price (strike price). This right is created for a specific time period, e.g. three, six or twelve months. The other party in the contract is known as the *writer* of the option. A *put* option is a contract to sell an underlying asset. An option contract creates an obligation for its writer to fulfill the holder's decision. For example, when a call option is exercised by the holder (that is the holder decides to buy the underlying asset), the writer of the option has to sell the underlying asset to the holder at the contract price. Call and Put are two types of options which can follow different styles. If the option can be exercised only at its expiration (i.e. the underlying asset can be purchased or sold only at the end of the life of the option) the option is referred to as an European style option. If it can be exercised on any date before its maturity, then the option is referred to as an American style option. These are two common styles of

options among many other complex options in the market such as Asian, Bermudan, and exotic options. Options can be written on numerous underlying assets, such as equity, precious metals, agricultural commodities, etc. We refer to those as real option pricing model.

The option pricing problem is to compute the price $F()$ at time t of a call or put option (on stock or other underlying asset of the option) of a given option style. This price F depends on various independent variables such as S, the current stock price; and various parameters such as T, the expiration time of the option contract; r, the risk-free interest rate; σ, the volatility of stock prices; and K, the strike price of the option.

Black and Scholes [Black and Scholes, 1973] and Merton [Merton, 1973] independently developed a theoretical model to price an option. The Black-Scholes-Merton model is a partial differential equation and can be solved for European options. For other styles of options where a closed form solution is not possible, numerical algorithms have been developed using techniques such as Monte Carlo [Boyle, 1977], binomial lattice [Cox et al., 1979] and fast Fourier transform [Carr and Madan, 1999]. In Chapter 2, we explain Black-Scholes-Merton and binomial lattice model in detail, since we are comparing the performance of our algorithm against them. The computational cost in all these techniques is quite high, and with parallel computing efforts [Barua et al., 2005; Rahmayil et al., 2004; Thulasiram and Thulasiraman, 2003; Thulasiram et al., 2001] major performance improvements have been achieved.

The Black-Scholes-Merton model assumes that the volatility of the underlying asset remains constant during the contract period of the option. This is one of the

limitations of the model. In a real market, volatility is dynamic and changes continuously. With complex models used to capture the real market conditions, it becomes difficult to find closed form solutions. This has lead researchers to consider modern numerical approaches including heuristic methods for solving the option pricing problem. One such technique is genetic programming(GP) [Yin et al., 2007]. GP is a population based search algorithm inspired by biological evolution. Given a set of high level statements and user defined tasks, GP automatically creates computer programs to solve the problem. It is a specialization of genetic algorithms where each individual is a computer program. Yin et al. [Yin et al., 2007] used GP to test the predictions of the Black-Scholes-Merton model against real market data. The authors relaxed the assumption of constant volatility and developed an adaptive GP algorithm.

Recently, bio-inspired and nature inspired algorithms [Brabazon and O'Neil, 2006] have been considered for financial modeling. These algorithms are inspired by techniques developed from behavior adapted by the insect societies for their own survival [Dorigo et al., 1996; Wedde et al., 2005]. Nature-inspired algorithms have gained prominence in real world optimization problems where the problem size is large, dynamic and complex. The option pricing problem falls into this category of problems. Nature-inspired algorithms have been used in many combinatorial optimization problems [Bullnheimer et al., 1999; Maniezzo and Colorni, 1999], and real world applications such as in mobile ad hoc networks [Caro et al., 2005; Wang et al., 2009]. Many of these problems are NP-Hard that researchers are interested in finding approximate solution in a reasonable amount of computational time. Ant Colony Optimization

(ACO) [Dorigo et al., 1996] is a nature-inspired algorithm inspired by real ants foraging for food. The ants find the shortest path from the nest to the food source.

In networking or combinatorial optimization problems the objective is to find the shortest path (distance) to travel to the destination, and therefore ACO is amenable to these problems. The original ACO [Dorigo et al., 1996] performs well in situations where the source and destination are known in advance. However, in option pricing the destination is unknown. Kumar et al. [Kumar et al., 2008, 2009] developed the first and unique ACO-based algorithm for the option pricing problem. Their pricing algorithm [Kumar et al., 2009] uses ants to predict the optimum profit achievable by exercising an option in a volatile market. The algorithm keeps track of the current optimum by evaluating the optimum state obtained so far. The computational domain is modeled as a directed acyclic graph and the ants travel along the edges of the graph. Since this is a very dynamic approach, the best node (represented by the profitable nodes in the graph) to exercise the option could be anywhere in the search space. One of the restrictions imposed on the algorithm is that the ants cannot travel backwards in time.

In the original ACO algorithm, the movement of ants is always forward until they reach the destination. Then, the ants move backward to the source. The movement is always in a straight line. Kumar et al.'s [Kumar et al., 2009] algorithm does not use backward ants, because there is no necessity to move back to the source. In the original ACO algorithm, there is no concept of time. In the option pricing problem, option exercise time is important and it is one of the parameters to be optimized for best option value. However, in applying ACO to the option pricing problem, the

algorithm [Kumar et al., 2009] does not keep track of the time. It only optimizes the option value. This is the one of the limitations of the algorithm. In option pricing, there are two objectives: (*i*) finding the best time to exercise the option; (*ii*) finding the best option value. The ACO algorithm [Kumar et al., 2009] considers the second objective.

In this research, we consider another technique also inspired by nature that tries to alleviate some of the problems discussed above. Particle Swarm Optimization (PSO) [Kennedy and Eberhart, 1995] is a population based algorithm inspired by birds flocking or fish schooling. PSO has been used widely in training neural networks [Chen et al., 2004, 2005; Jha et al., 2009b; Nenortaitė, 2007] in financial applications such as time series forecasting and trading. In option pricing, Lee et al. [Lee et al., 2007] have used PSO for volatility estimation.

Jha et al. [Jha et al., 2009a] did a feasibility study by developing a simple algorithm using PSO to predict the maximum early profit by exercising an option. This algorithm assumes constant volatility, and was experimented in MATLAB for small test cases to price an European option. In this work, volatility is calculated from a simple variance equation provided in the MATLAB toolbox with a normal distribution of the asset prices. The experiments showed that PSO produces promising results, and converges to a near optimal solution.

In this research, we design, develop, and implement a sequential PSO based algorithm for the option pricing problem using the basic principles of PSO. We make two major improvements to our previous [Jha et al., 2009a] algorithm: (*i*) incorporate variable volatility (Section 4.1.1), and (*ii*) capture both the profit and time

(Figure 4.1).

One of the hard problems is to map the PSO to the option pricing problem. In Section 4.1, we explain how this mapping can be done. Also, the algorithm is applicable for both European and American options unlike Jha et al.'s work which is applicable to the simple European option with constant volatility.

The rest of this thesis is organized as follows: Chapter 2 presents option pricing techniques. Chapter 3 discusses Particle Swarm Optimization in detail. In Chapter 4, we show the mapping of PSO for option pricing, and present our sequential PSO algorithm followed by a parallel algorithm. In Chapter 5, we present experimental results and discussion. Finally, in Chapter 6 we conclude the thesis with direction for future work.

Chapter 2

Option Pricing: Models and Techniques

In this chapter, we describe Black-Scholes-Merton model, binomial lattice model, ACO and PSO based algorithms available to price an option.

2.1 Black-Scholes-Merton Option Pricing Model

Black-Scholes-Merton [Black and Scholes, 1973; Merton, 1973] model is an early mathematical model for simple options that can be solved for closed form solution. Two of the major limitations of this model are: (i) the model derivation is based on the assumption that volatility of the underlying asset remains constant during the life of an option, and (ii) the model can handle only simple European options.

The Black-Scholes-Merton formula, see for example [Hull, 2008], for the option price at time 0 of a European call (C) on a non-dividend-paying stock and an Euro-

pean put option (P) on a non-dividend-paying stock are

$$C = S_0 \times N(d_1) - K \times e^{-rT} \times N(d_1) \tag{2.1}$$

$$P = K \times e^{-rT} \times N(-d_2) - S_0 \times N(-d_1) \tag{2.2}$$

respectively, where

$$d_1 = \frac{ln(\frac{S_0}{K}) + (r + \frac{\sigma^2}{2}) \times T}{\sigma \times \sqrt{T}} \tag{2.3}$$

$$d_2 = d_1 - \sigma \times \sqrt{T} \tag{2.4}$$

Here, $N(x)$ is the cumulative probability distribution function for a standardized normal distribution. C and P are the European call and put option values. S_0, stock price at time zero; T, the expiration time of the option contract; r, the risk-free interest rate; σ, the volatility of stock prices; and K, the strike price of the option.

2.2 Binomial Lattice Option Pricing Model

The binomial lattice [Cox et al., 1979] is discrete time option pricing model. It is flexible, popular, and easy to understand and implement. One of the major advantages of this model is that it can be used for pricing American options. This model uses a binomial tree structure for asset price movements. Figure 2.1 is an example of one-step binomial tree, where S_0 is the stock price at the beginning of the contract period. The stock price S_0 can go up (by a multiplicative factor u) to uS_0 with probability q, or can decrease to dS_0 with complementary probability $(1 - q)$ at the end

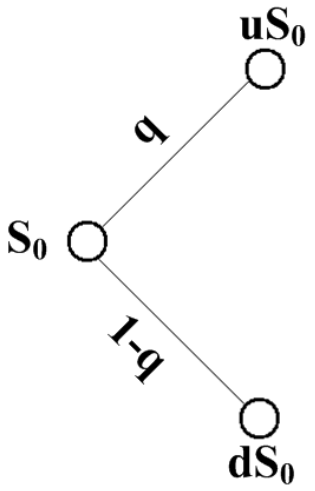

Figure 2.1: one-step binomial tree

of the period. Here u and d are the factors by which the stock price goes up or down respectively, and are given by $u = e^{\sigma\sqrt{\Delta t}}$ and $d = e^{-\sigma\sqrt{\Delta t}}$ where σ is volatility.

The pay-off from the derivative is f_u, as the price S moves up and is given by

$$f_u = \begin{cases} max(S-K,0) & \text{for a call option} \\ max(K-S,0) & \text{for a put option} \end{cases} \tag{2.5}$$

Similarly, as S moves down the pay-off f_d is computed. For the one-step binomial model, the option value at the current node is computed using the pay-off from the two possibilities of price movements in future in a binomial tree and is given by [Cox et al., 1979],

$$f = e^{-r\Delta t}(qf_u + (1-q)f_d), \tag{2.6}$$

where

$$q = \frac{e^{r\Delta t} - d}{u - d}, u - d \neq 0 \tag{2.7}$$

and $e^{-r\Delta t}$ is the discounting factor used to compute option value f at the current time t. In other words, option value is the discounted value of weighted sum of the future pay-offs due to up and down movement of the underlying asset.

To capture asset price movement closely, the contract period is divided into N steps so that when time step $\Delta t(= \frac{T}{N})$ tends to zero, the discrete time binomial lattice will approach the continuous time Black-Scholes-Merton model.

For the N-step binomial model [Thulasiram et al., 2001], the stock price movement of an underlying asset is described by a strict multiplicative binomial process over successive periods. The time between start and maturity period of a contract is divided into N intervals. Each interval will have certain number of nodes. Each node in a tree represents a possible price of stock at a particular time. Larger value of N is preferable as it allows to capture wider possible stock prices.

The N-step binomial tree exhibits high degree of built-in concurrency which can be exploited for parallel implementation. Many algorithms have been developed using this technique for the option pricing problem (for example [Chalasani et al., 1999; Huang and Thulasiram, 2005; Thulasiram et al., 2001]). The major advantage of this technique is the intuitiveness of the method. The disadvantages are in the slower convergence of the results and computational cost.

2.2.1 A Numerical Example

In this example, we compute option value for a call option whose underlying asset is a stock. Figure 2.2 is a three-step binomial tree for an option with 9 months contract period where we assume risk neutral probabilities stay the same through out

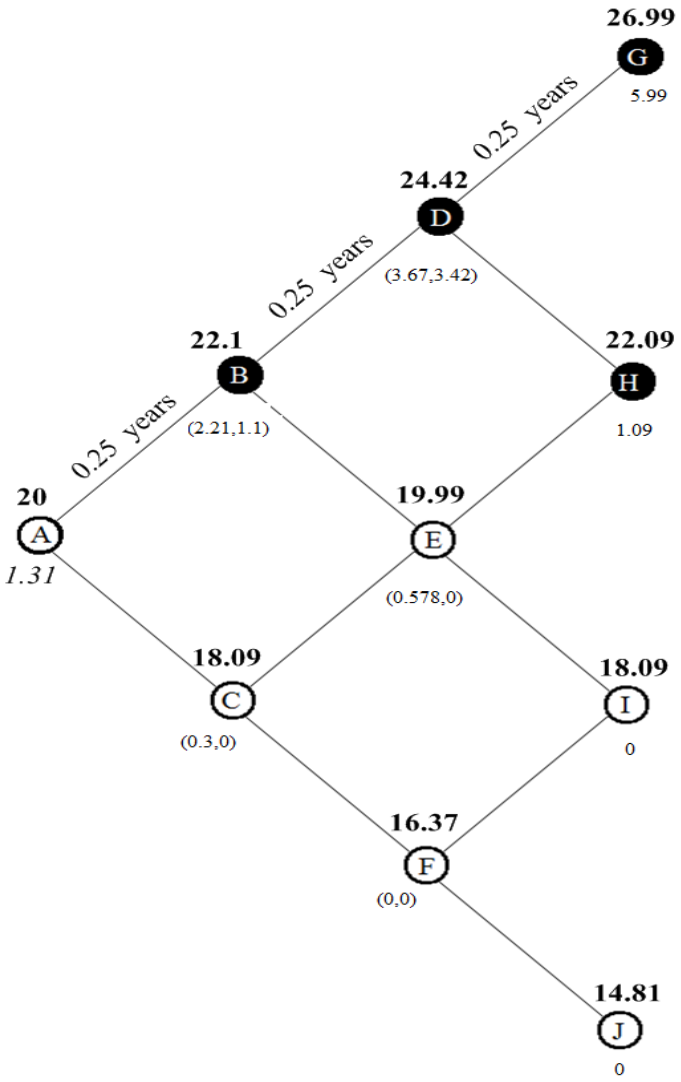

Figure 2.2: Three-step binomial tree

the process. Node A represents the current time where stock price $S_0 = \$20$. Nodes B and C are the intermediate nodes in the contract period at the end of 3 months. Similarly, nodes D, E, and F are the intermediate nodes in the contract period at the end of 6 months. The leaf nodes G, H, I, and J are the possible terminal points that the stock will reach at the end of 9 months. The intervals are set to be equal for

simplicity. In Figure 2.2, the values below each node are the (calculated option value and local pay-off), and the values above the node are the stock price. The option values and stock prices are computed for $\sigma = 0.2$, $r = 0.05$, $K = \$21$, $S_0 = \$20$, $T = 9$ months, and $\Delta t = 3$ months. The value of u and d are 1.1052 and 0.9048 respectively, and are calculated using the equations presented in Section 2.2.

At G, H, I and J (the leaf nodes) the option prices calculated are the local pay-off for the call option. They are calculated from the strike price and speculated stock price. Therefore, at G, for the local pay-off for the call is $= max(stock\ price - strike\ price, 0.0) = max(\$26.99 - \$21, 0.0) = \$5.99$. The local pay-off is the value of the option as soon as the node is reached. At nodes I and J, since the $stock\ price < strike\ price$, the option value for a call at these nodes is \$0.

To calculate an option value at intermediate nodes, we first find the value of q using Equation 2.7. For this example, the value of q is 0.538. Then, we calculate option value at a node using future prices and discounted pay-off as given in Equation 2.6.

For European style option, option value is calculated at time 0 (root node). For this example, the option value computed at root node, node A, is 1.31.

In American style option, we search for the best intermediate node to exercise an option since it is allowed to exercise an option any time before a maturity date. In the current example, an American option could be exercised before expiry either at the end of 3 or 6 months. In this example, nodes C, E or F are not desirable since local pay-off is zero; however, for example at E since the computed option value (0.578) is greater than the local pay-off which is zero, waiting for one more period may bring a positive option value at node E; similar observations can be made at node

C; at nodes B and D the computed option values are higher than the corresponding nodes in C, E, and F and therefore desirable. At node B, the option value is greater than local pay-off ($= 22.1 - 21 = 1.1$), thus waiting for one more period might yield a profitable solution. Similarly, at node D, option value (3.67) is greater than local pay-off ($= 24.42 - 21 = 3.42$), thus waiting for one more period might yield a profitable solution. Between nodes B and D, D is profitable and also since there are no intermediate nodes after D, D is preferable to exercise an American option.

2.3 Heuristic Approaches for Option Pricing

In this section, we describe some heuristic techniques available to price an option.

Artificial neural networks have been used for hedging [Hutchinson et al., 1994] and time series prediction [Chen et al., 2006] in finance applications. Genetic programming (GP) [Chen et al., 1999; Chidambaran et al., 1999] approaches have been used to price options in the literature. Keber and Schuster [Keber and Schuster, 2003] used ideas from genetic programming and ant systems called generalized ant programming (GAP) to derive formulas for calculating implied volatility of the underlying asset of an American put option. Implied volatility is the volatility of an asset that is calculated using the value of the option. There are no closed form solutions for calculating implied volatilities. Therefore, analytical approximations [Bharadia et al., 1995, 1996; Chance, 1996; Keber, 1999] are considered. Keber and Schuster [Keber and Schuster, 2003] showed through experimentation that their formula produces accurate approximation results and outperforms other approximations described in the literature.

Yin et al. [Yin et al., 2007] used genetic programming to test the predictions of the

Black-Scholes-Merton model against real market data. The mutation and crossover probability rates are fixed, in general, in the GP technique. Yin et al. [Yin et al., 2007] dynamically altered the mutation and crossover rates in each GP run. They claimed that this adaptive algorithm captures the market in real time and produces better implied volatility approximations.

We briefly describe ACO and PSO in the rest of this chapter, and elaborate on PSO in the next chapter.

2.3.1 Ant Colony Optimization (ACO) for option pricing

In option pricing literature, most of the work using heuristics has been to develop approximation formulas for the implied volatility. Kumar et al. [Kumar et al., 2009] first developed an ant colony optimization (ACO) based algorithm to price options. Their dynamic iterative algorithm inherently captures the market volatility during the life of the option. Initially, all ants start searching the solution space from an initial node. The objective of the ants is to find the best node (profitable node) to exercise an option. Ants move towards the global best node by choosing a path to the next node that has high concentration of the pheromone. After a few iterations, more ants are injected at the best node to explore the solution space further. Kumar et al. [Kumar et al., 2009] showed that the algorithm performs better than binomial-lattice algorithm.

The advantage of dynamic iterative algorithm is that it works for volatile market. That is, the algorithm captures volatility of the underlying asset along the path. The dynamic iterative algorithm incurs additional computational cost due to introduction

of new ants and in short interrupts in their movement. The main cause for the cost is in updating ants data structures. There was one restriction in the algorithm-the ants were not allowed to travel backwards in time and therefore the algorithm does not optimize the time parameter. This has significantly impacted the results in some test cases where it could capture only sub-optimum solution.

2.3.2 Particle Swarm Optimization (PSO) for Option Pricing

Particle Swarm Optimization (PSO) unlike ACO, tries to find the best node in a graph or search space rather than the best path. Since ACO works well when the destination node is known in advance (such as in network applications), the original ACO had to be modified to suit the financial application since the destination is unknown in the option pricing problem. In PSO, destination information is not required, and the particles are allowed to fly in multiple directions looking for optimal solution. It is therefore, very amenable to option pricing. To our knowledge there is only one work in the literature [Lee et al., 2007] where PSO has been used to calculate implied volatilities. Jha et al. [Jha et al., 2009a] did the first feasibility study in using PSO to price European-style options. The authors implemented the algorithm using the PSO toolbox developed by Birge [Birge, 2003] which is available in MATLAB. The experiments were done with a small population size (number of particles), and the results deviated from the Black-Scholes-Merton model. The goal of this work was to determine the feasibility of using PSO to price options.

Chapter 3

Particle Swarm Optimization (PSO)

Particle Swarm Optimization [Kennedy and Eberhart, 1995] is a population based heuristic optimization algorithm inspired by social behavior of birds flocking or fish schooling.

Consider the scenario of birds looking for food. The birds first search their own neighborhood for the food source. At the end of each time step or iteration the birds decide on a location that might lead to a food source. The birds then compare their solution with other birds solutions and move closer to the birds that are closest to the location of the food source. Each bird (a particle) is a potential solution in the search space. This concept is formulated as PSO algorithm.

Each particle is treated as a point in a D-dimensional space. Initially, N particles are uniformly distributed in the solution space. The particles in PSO fly through the search space with a certain velocity, and change their position dynamically in the

hope of reaching the food source, the destination. Therefore, position and velocity are two important parameters in the PSO algorithm.

Each particle keeps track of the best position it has encountered during its travel, and the best position traveled by the swarm of particles. The best position traveled by a particle is called the *local best position*, and the best position traveled by the swarm is called the *global best position*. At the end of each iteration, the particles calculate their next velocity, and update their positions based on the calculated velocity.

3.1 PSO Algorithm

There are fours steps in PSO algorithm: initialization, position update, evaluation, and termination. These steps are explained as follows:

Initialization

The first step is to initialize N particles randomly in a solution space. Equation 3.1 describes initial position, and Equation 3.2 gives the initial velocity.

$$\overrightarrow{X_0^i} = \overrightarrow{X_{min}} + r_1 \times (\overrightarrow{X_{max}} - \overrightarrow{X_{min}}) \tag{3.1}$$

$$\overrightarrow{V_0^i} = \overrightarrow{X^i} \tag{3.2}$$

For a particle i, $\overrightarrow{X_0^i}$ is the initial position, $\overrightarrow{X_{min}}$ is the minimum position, $\overrightarrow{X_{max}}$ is the maximum position, r_1 is a uniform random number within the range [0,1], and $\overrightarrow{V_0^i}$ is the initial velocity.

Position Update

During the execution of the algorithm, each particle i monitors four values: its current position ($\overrightarrow{X_t^i}$), the best position it reached in previous iterations ($\overrightarrow{P_t^i}$), its flying velocity ($\overrightarrow{V^i}_t$), and the swarm best position ($\overrightarrow{P^g}_t$) at time t. These four values are represented in vectors as: $\overrightarrow{P_t^i} = (p_t^{i1}, p_t^{i2}, p_t^{i3}, \ldots, p_t^{iD})$; $\overrightarrow{V_t^i} = (v_t^{i1}, v_t^{i2}, v_t^{i3}, \ldots, v_t^{iD})$; $\overrightarrow{P_t^g} = (p_t^{g1}, p_t^{g2}, p_t^{g3}, \ldots, v_t^{gD})$; $\overrightarrow{X_t^i} = (x_t^{i1}, x_t^{i2}, x_t^{i3}, \ldots, x_t^{iD})$ and are used to calculate the next velocity given by Kennedy and Eberhart [1995]:

$$\overrightarrow{V_{t+1}^i} = \overrightarrow{V_t^i} + c_1 \times r_1 \times (\overrightarrow{P_t^i} - \overrightarrow{X_t^i}) + c_2 \times r_2 \times (\overrightarrow{P_t^g} - \overrightarrow{X_t^i}) \tag{3.3}$$

In Equation 3.3, r_1 and r_2 are two uniform random numbers within the range $[0, 1]$, c_1 is self or local confident factor, and c_2 is swarm or global confident factor. The velocity Equation 3.3 was modified by Shi and Eberhart [Shi and Eberhart, 1998] by introducing a new parameter called inertia weight ω to balance the trade off between global and local search during optimization process. The consequent velocity equation is given in Equation 3.6. They have found a significant performance improvement of the PSO method with a linearly varying inertia weight (ω) over the iterations. The mathematical representation of this concept is given by Equation 3.4

$$\omega = (\omega_1 - \omega_2) \times \frac{(MAXITER - iter)}{MAXITER} + \omega_2 \tag{3.4}$$

where ω_1 and ω_2 are initial and final values of the inertia weight respectively, $iter$ is current iteration number, and $MAXITER$ is the maximum number of allowable iterations, a user parameter.

Through empirical studies, Shi and Eberhart [Shi and Eberhart, 1998] have observed that the optimal solution can be improved by varying the value of ω from 0.9

at the beginning of search to 0.4 at the end of search for most problems, which is also called as time-varying inertia weight.

The new position equation, Equation 3.5, and new velocity equation, Equation 3.6, proposed by Shi and Eberhart are given below. These equations are used to update the particles next position after it calculates velocity.

$$\overrightarrow{X^i_{t+1}} = \overrightarrow{X^i_t} + \overrightarrow{V^i_{t+1}} \tag{3.5}$$

$$\overrightarrow{V^i_{t+1}} = \omega \times \overrightarrow{V^i_t} + c_1 \times r_1 \times (\overrightarrow{P^i_t} - \overrightarrow{X^i_t}) + c_2 \times r_2 \times (\overrightarrow{P^g_t} - \overrightarrow{X^i_t}) \tag{3.6}$$

We construct an equation similar to this velocity equation with appropriate parameters to track changes in the asset price.

Furthermore, Eberhart and Shi [Eberhart and Shi, 2001] in another work stated that the time-varying inertia weight is not very effective for tracking dynamic systems, even though it is effective in optimizing static problems. Also, most real-world applications are identified as nonlinear dynamic systems. Considering the dynamic nature of real-world applications, Eberhart and Shi [Eberhart and Shi, 2001] proposed a random inertia weight factor for tracking dynamic systems, which is given as:

$$\omega = 0.5 + \frac{rand}{2} \tag{3.7}$$

where "rand" is a uniform random number within the range [0,1].

ω is an important parameter in PSO. In option pricing, random part of ω considers the random behavior of market over a period of time.

In velocity Equation 3.6, a particle's movement is limited in order to control its trajectories. Control on a particle's trajectory is used to prevent a particle's explosion. Particle explosion means a particle moving out of the solution space. That is, we limit the speed of the particles to some constant $\overrightarrow{V_{max}}$ value. However, limiting the movement of a particle has disadvantages. For example, in option pricing problem, the value of underlying assets can rise or fall to any value over a period of time. Clerc and Kennedy [Clerc and Kennedy, 2002] suggested a constricted version of PSO. The constricted version of PSO prevents explosive behavior of the algorithm without a need for a velocity bounding constant $\overrightarrow{V_{max}}$. The constricted Clerc and Kennedy's [Clerc and Kennedy, 2002] velocity equation is given as

$$\vec{V_{t+1}^{i}} = \chi \times [\vec{V_t^i} + \phi_1 \times (\vec{P_t^i} - \vec{X_t^i}) + \phi_2 \times (\vec{P_t^g} - \vec{X_t^i})] \tag{3.8}$$

where χ is a constricted coefficient, which is calculated using Equation 3.9 given below:

$$\chi = \begin{cases} \frac{2\beta}{\phi_2 + \sqrt{\phi^2 - 4\phi}} & \text{if } \phi > 4; \\ \beta & \text{else.} \end{cases} \tag{3.9}$$

Essentially, ϕ_1 and ϕ_2 are two positive random numbers, and are calculated as: $\phi_1 = r_1 \times \frac{\phi_{max}}{2}$, $\phi_2 = r_2 \times \frac{\phi_{max}}{2}$; r_1 and r_2 are two uniform random numbers in closed interval [0,1]; ϕ is constant number calculated using $\phi = \phi_1 + \phi_2$, and $\beta \in (0, 1)$.

Comparing Clerc and Kennedy velocity equation (Equation 3.8) with Shi and Eberhart (Equation 3.6), we can see that c_1 is equivalent to ϕ_1, which is equal to $r_1 \times \frac{\phi_{max}}{2}$, and c_2 equivalent to ϕ_2, which is equal to $r_2 \times \frac{\phi_{max}}{2}$. Therefore, one can infer that ϕ_{max} is equivalent to $max(2 \times c_1, 2 \times c_2)$. In Clerc and Kennedy velocity

Suggested by	ω	c_1	c_2	N
Kennedy and Eberhart [Kennedy and Eberhart, 1995]	1	2	2	15-40
Shi and Eberhart [Shi and Eberhart, 1998]	1.2-0.9	2	2	15-40
Clerc and Kennedy [Clerc and Kennedy, 2002]	1-0	2.05	2.05	—
Trelea [Trelea, 2003]	0.6	1.7	1.7	30

Table 3.1: PSO parameters selection

equation χ with the help of β considers volatility. In this thesis, we consider Shi and Eberhart's velocity equation.

Table 3.1 captures the parametric conditions used by various authors in the past.

Evaluate fitness

In this step, all particles evaluate a fitness function (for our algorithm fitness function is discussed is Section 2.2) at their respective positions. Local and global updates are done if particles find better fitness values.

Termination

The search stops in either of two conditions: a particle reaches the preset target accuracy or satisfies the preset maximum number of iterations. The preset target accuracy depends on the type of optimization problem. In our case, target profit is preset value, and maximum number of iterations is the termination condition.

Chapter 4

PSO for Option Pricing

One of the important research problems in computational finance is identifying the best time to exercise a given option while maximizing the profit of the option. Approaches described earlier may achieve this objective, however, at a higher computational cost. The focus of this thesis is to design an algorithm that is inspired from natural world such as school of fish, flock of birds or in general swarm. This algorithm is expected to compute optimal option values in the given solution space at a reasonable amount of time which is better than previous approaches.

4.1 A Sequential Option Pricing Algorithm using PSO (SPSO)

In this thesis, we consider a popular nature inspired algorithm, Particle Swarm Optimization (PSO). The first focus of our research is to design and develop a sequential algorithm for the option pricing problem using basic principles of PSO. Jha

et al. [Jha et al., 2009a] algorithm did a feasibility study of using PSO for option pricing with MATLAB. We redesign and improve this algorithm to incorporate larger population size (N), terminating conditions ($MAXITER$ or target profit), volatility (both constant and variable) parameters for an underlying asset.

Jha et al. [Jha et al., 2009a] used the toolbox available in MATLAB where volatility is calculated from a simple variance equation provided in MATLAB with an assumed distribution of the asset prices. In this work, we are developing an algorithm to price options using basic principles of PSO. Our algorithm will work for both constant and variable volatilities.

Two major improvements over Jha et al.'s [Jha et al., 2009a] algorithm are: (i) in incorporation of volatility (Section 4.1.1), and (ii) structure of a particle to capture both the option value (profit) and time (Figure 4.1). In our work, one of the hard problems is to map the PSO to the option pricing problem. We explain below how this mapping can be done. Also, the algorithm is applicable for both European and American options unlike Jha et al.'s work which is applicable to the simple European option with constant volatility.

4.1.1 Incorporating volatility in SPSO

First we discuss how we theoretically incorporate volatility in PSO velocity equations. We consider both velocity equations (Shi and Eberhart(Equation 3.6) and Clerc and Kennedy (Equation 3.8)) to incorporate volatilities for option pricing. In both velocity equations, we consider constant and dynamic volatility. Note that the velocity equation calculates change in asset price from one iteration to another over

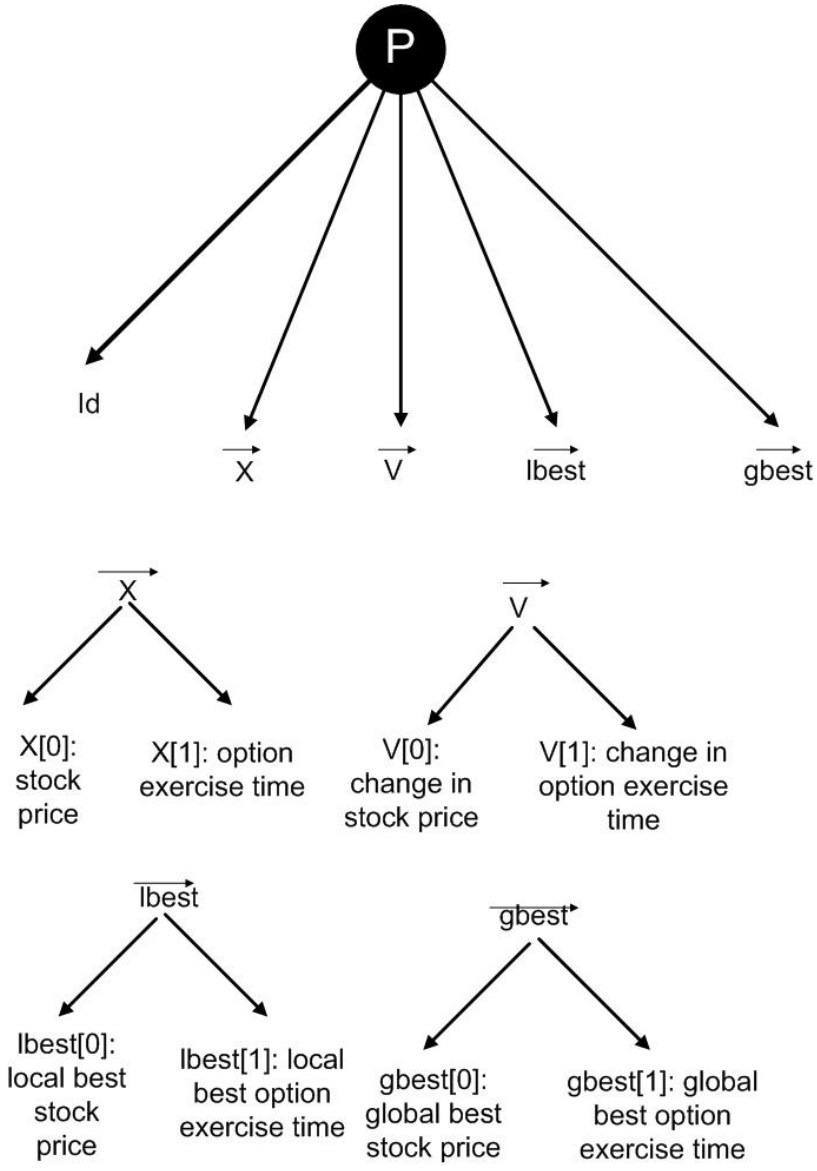

Figure 4.1: Structure of a particle in PSO

a period of time, and change depends on the volatility of an asset price; the position vector $\vec{X}$ represents asset price.

The parameter ω in Eberhart and Shi (Equation 3.7) is a dynamic inertia weight that stabilizes the system to balance local and global search in PSO. For option

pricing, random part of ω acts as volatility of the underlying asset. In Equation 3.6 and Equation 3.8, ω and β (through χ) act as the volatility component respectively.

Eberhart and Shi suggested an improved parameter ω in Equation 3.7. For our purpose, we replace the random term in Equation 3.7 with volatility σ, which in essence represents the random behavior of an asset price movement. That is, the parameter ω can be written as:

$$\omega = 0.5 + \sigma \tag{4.1}$$

By assigning a value to σ, we can simulate particles evolution with constant volatility. Also, by randomly changing the value of σ for each particle's evolution, we can simulate real market scenario.

Though it is not implemented for this thesis if we want to use velocity equation (Equation 3.8), suggested by Clerc and Kennedy, the parameter β can be seen equivalent to volatility σ.

4.1.2 Mapping PSO for Option Pricing

In mapping the PSO to the option pricing problem, a particle in our algorithm is defined by five parameters: id, position vector ($\vec{X}$), velocity vector ($\vec{V}$), local best position ($\vec{l}_{best}$) and global best position ($\vec{g}_{best}$). $\vec{X}$ corresponds to the price and $\vec{V}$ corresponds to the change in price. $\overrightarrow{X_{max}}$ and $\overrightarrow{X_{min}}$ correspond to the boundaries of the solution space. Similarly, previous best position $\overrightarrow{P_t^i}$ in PSO is a particle's best ($\overrightarrow{l}^i_{best}$) in terms of option value (profit) and option exercise time; and $\overrightarrow{P_t^g}$ in the PSO is the overall best ($\overrightarrow{g}_{best}$) among particles in the option pricing solution space. The

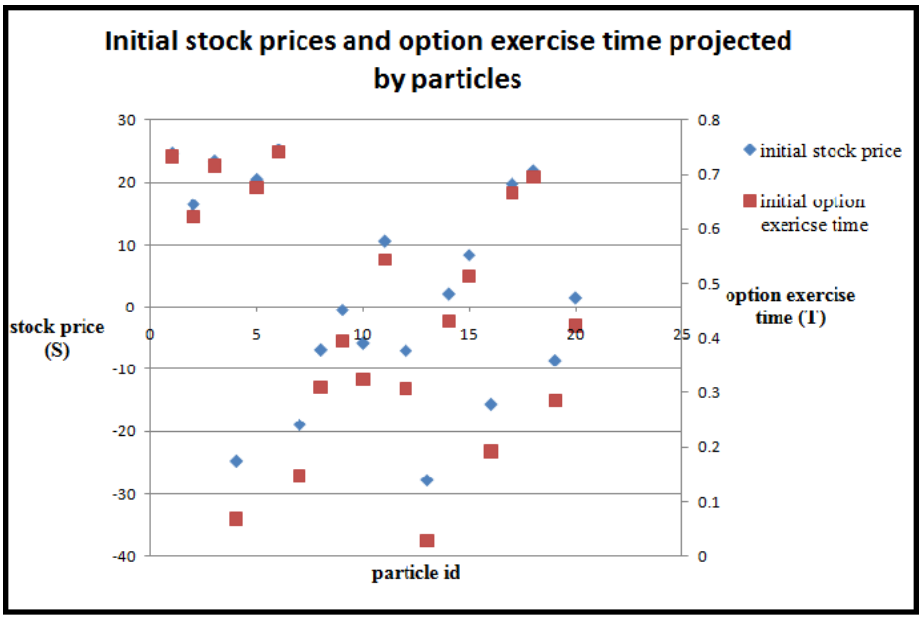

Figure 4.2: Initial particles position

initial particle position and velocity are already defined in Equations 3.1 and 3.2 respectively.

A particle is used to compute option value and capture the time when this optimum option value occurs. Therefore, each position vector ($\vec{X}$) has two dimensions. The first dimension ($X[0]$) is for stock price, and the second dimension ($X[1]$) is the exercise time. Similarly, there are two dimensions for $\vec{V}$, $\vec{l}_{best}$ and $\vec{g}_{best}$, $\overrightarrow{X_{max}}$ and $\overrightarrow{X_{min}}$.

Initially, the particles are distributed randomly in the solution space using Equation 3.1. Figure 4.2 is an example of initial distributions of particles. In this figure, stock prices and option exercise time projected by particles are shown. The initial velocity for the i^{th} particle is calculated using Equation 3.2. The particles fly through the search space with a certain velocity, and change their position dynamically until

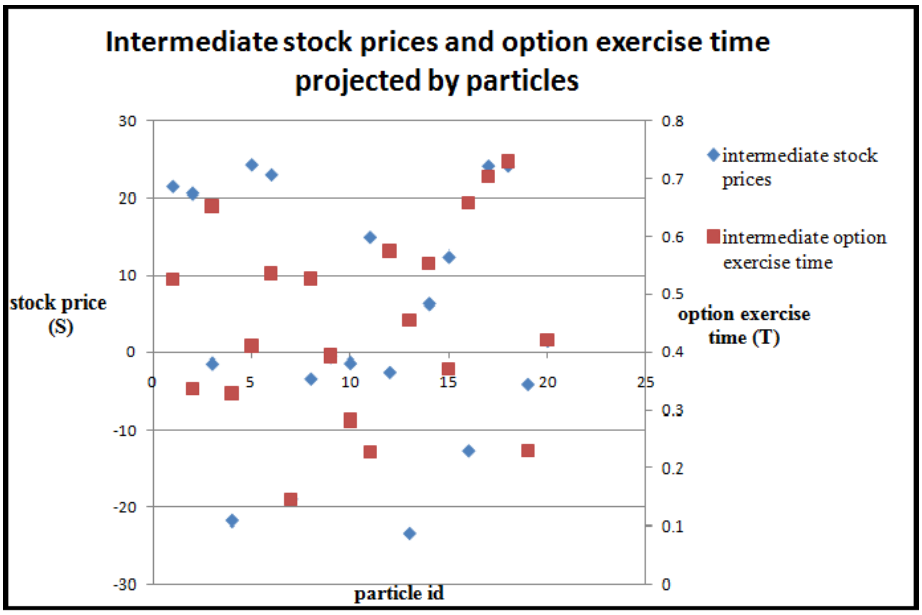

Figure 4.3: Intermediate particles position

the termination condition is satisfied. In the current implementation the termination condition is set to a predetermined number of iterations. Figure 4.3 is an example of intermediate positions of particles. In this figure, intermediate stock prices and option exercise time projected by particles are shown.

Figure 4.4 is an example of final positions of particles. In this figure, final stock prices, option value and option exercise time projected by particles are shown. The position and velocity updates by a particle i are calculated as follows:

$$\overrightarrow{X^i_{t+1}[k]} = \overrightarrow{X^i_t[k]} + \overrightarrow{V^i_{t+1}[k]}; k = 0, 1. \tag{4.2}$$

$$\overrightarrow{V^i_{t+1}[k]} = \omega \times \overrightarrow{V^i_t[k]} + c_1 \times r_1 \times (\overrightarrow{l^i_{best}[k]} - \overrightarrow{X^i_t[k]}) + c_2 \times r_2 \times (\overrightarrow{g_{best}[k]} - \overrightarrow{X^i_t[k]}); k = 0, 1. \tag{4.3}$$

$(\overrightarrow{X^i})$ and $(\overrightarrow{V^i})$ are a particle's current position and flying velocity, respectively.

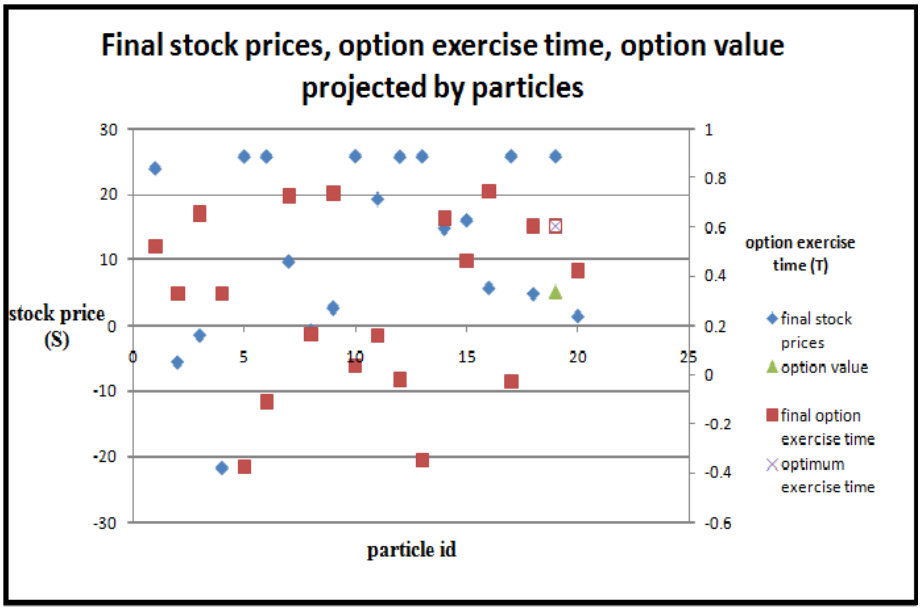

Figure 4.4: Final particles position

$\overrightarrow{l^i_{best}[k]}$ and $\overrightarrow{g_{best}[k]}$ are initially assigned based on the random initialization of the particles.

There are four possible scenarios to consider for a particle to update its $\overrightarrow{l^i_{best}[k]}$ and $\overrightarrow{g_{best}[k]}$ after it moved to a new position. Figures 4.5- 4.8 demonstrate these scenarios for a call option. In these figures, "X" represents local best position, and a circle represents a new position of a particle.

Scenario I :

A particle flies to a new position P_1 (node), and checks if that new position represents better profit (higher local pay-off; that is, higher stock price for a call option) and represents an earlier time than the local best (price and time) obtained so far for that particle (see Figure 4.5). If so, the particle updates its local best position as shown in Figure 4.6. Recall that we are demonstrating

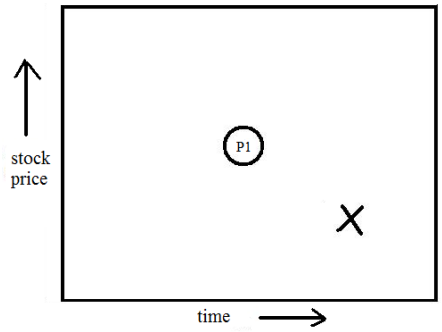

Figure 4.5: A particle movement for a call option Scenario I

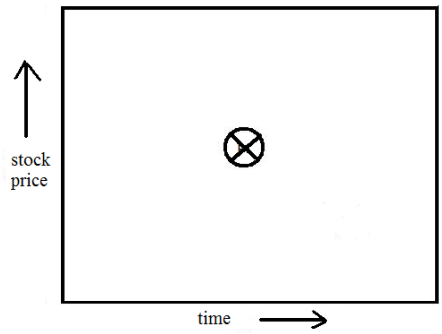

Figure 4.6: Local best update

the particles movement for a call option. Similar arguments can be extended to a put option.

Scenario II :

A particle flies to a new position P_2, and checks if that new position represents better profit (higher stock price for a call option) and represents a later (future) time than the local best position (price and time) obtained so far (see Figure 4.7) for that particle. If so, the particle does the following:

- the option price at the new position (future time) is discounted using

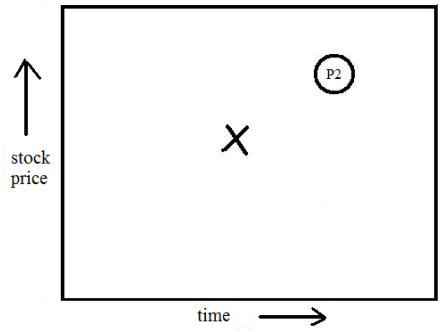

Figure 4.7: A particle movement for a call option Scenario II

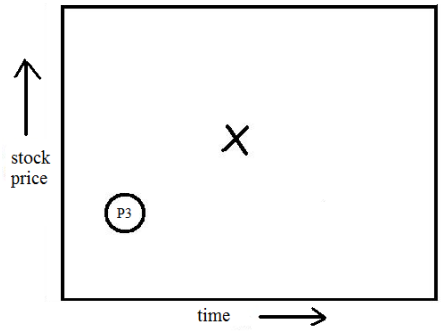

Figure 4.8: A particle movement for a call option Scenario III

$(X^i[0]-K)\times\exp(-r\times(X^i[1]-l^i_{best}[1]\ ))$, (where K is the strike price and r is the risk free interest rate), to find its value at the local best position. Recall that $X[1]$ and $l_{best}[1]$ correspond to time.

- if this discounted value is greater than the current local best option price, then the particle flies to the new position (that is, waiting for the future time is beneficial), and updates its local best position with the newly computed option price and exercise time. Otherwise, the local best remains as it is.

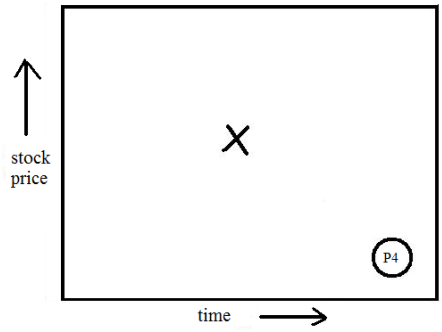

Figure 4.9: A particle movement for a call option Scenario IV

Scenario III :

A particle flies to a new position P_3, and in that position if profit is less (i.e. lower local pay-off; that is lower stock price for a call option) and time in this new position represents an earlier time, then the particle does the following. Though the local pay-off is less at this new position, the time represented by this position is an earlier time than the local best time (see Figure 4.8). There is a chance that the newly found node may move to profitable position (with higher local pay-off) at later time (iterations) than the local best position. In this case, the particle needs to search the surrounding neighborhood. This is to prevent stagnation at one location. The particle does the following:

- the future option value is discounted using $(X^i[0]-K)\times\exp(-r\times(l^i_{best}[1]-X^i[1]))$ to calculate the current option value.
- if the discounted future option value is greater than the current local best option price, then particle updates its local best position. Otherwise, the local best remains as it is.

Scenario IV :

The last case where there is less profit (i.e. smaller local pay-off) at a later date (see Figure 4.9) in time, and is not advantageous because (a) particles may have already found a better position which is profitable and corresponds to an earlier time (b) particles are close to maturity date. In this case, a particle performs no update on its local best position.

After each iteration, the best of all the local best ($\vec{l}^{i}_{best}$) is calculated in the same was as the above four cases in comparison with $\vec{g}_{best}$. If ($\vec{l}^{i}_{best}$) is better than current $\vec{g}_{best}$ (global best), then $\vec{g}_{best}$ will be updated. The update in local and global best positions continues until termination condition is met.

4.1.3 A sequential PSO-based algorithm for Option Pricing

The pseudo code for a sequential PSO-based option pricing algorithm is given below:

Initialize PSO parameters: c_1, c_2, ω, $\vec{V}^{i}_{max}$, $\vec{V}^{i}_{min}$, $\vec{X}^{i}_{max}$, $\vec{X}^{i}_{min}$

Initialize option pricing variables and parameters: S, K, r, σ, T

for $i = 1$ to N **do**

Initialize $\vec{X}^{i}$ by using Equation 3.1

Initialize $\vec{V}^{i}$ by using Equation 3.2

$\vec{l}^{i}_{best} = \vec{X}^{i}$

end for

$\vec{g}_{best} = \max \vec{l}^{1}_{best}, \vec{l}^{2}_{best}, \ldots, \vec{l}^{N}_{best}$

for $Itr = 1$ to $MAXITER$ **do**

for $i = 1$ to N **do**

Calculate the new position ($\vec{X}^i_{Itr}$) and velocity ($\vec{V}^i_{Itr}$) by using Equations 4.2 and 4.3 respectively.

Check boundary conditions: If a particle flies out of boundary (solution space i.e. $\vec{X}^i_{max}$ or $\vec{X}^i_{min}$), then re-calculate new velocity, and check boundary conditions one more time. If a particle flies out of solution space with re-calculated velocity, then penalize the particle with some random velocity.

Update the local best position by considering the four scenarios discussed in section 4.1.2.

end for

$\vec{g}_{best} = \max \vec{l}^1_{best}, \vec{l}^2_{best}, \ldots, \vec{l}^N_{best}$

Update $\vec{g}_{best}$ information in all particles

end for

Print $\vec{g}_{best}$

4.2 Parallel PSO Algorithms

There are two parallel PSO algorithms: synchronous and asynchronous. Schutte et al. [Schutte et al., 2004] designed the first general synchronous parallel PSO algorithm. Subsequently, asynchronous PSO algorithm is developed for certain applications [Koh-II et al., 2006; Li et al., 2007; Venter and Sobieszczanski-Sobieski, 2006] but with little performance increase. Parallel computing is also essential in the option pricing problem, due to the need for real time pricing to beat the competition in the market. Therefore, we develop a synchronous parallel version of the proposed

sequential algorithm which is suitable on various parallel architectures.

4.2.1 A parallel synchronous option pricing algorithm using PSO

This section describes the parallel synchronous algorithm. This is based on the sequential algorithm discussed in Section 4.1.3.

We design our parallel algorithm for three different architectures.

4.2.2 Shared/distributed/hybrid architecture

A distributed memory machine [Grama et al., 2003] is a multicomputer with disjoint address space. Each processor has access to its own local memory. Communication between processors is through message passing. The standard message passing interface used is MPI. A shared memory machine [Grama et al., 2003] is a multiprocessor with a global memory space used by all processors. Communication between processors is through shared variables. OpenMP is the standard parallel programming language used in shared memory machines. A hybrid memory machine is a cluster of multiprocessors. These machines support features of distributed and shared memory machines. Each node has p processors sharing a global address space. There may be H such nodes interconnected by a high speed Ethernet connection. Communication between nodes is through message passing while communication within a node is through shared variables. Therefore, OpenMP maybe used within a node and MPI between nodes. An MPI program can be converted to a MPI/OpenMP program to run on hybrid machines. These machines give the benefit of both types

of architectures.

The Helium cluster available in the Department of Computer Science at University of Manitoba is a hybrid machine that supports hybrid implementation, pure OpenMP implementation and pure MPI implementations. Each node contains eight dual-core homogeneous processors (all processors are of same characteristics) supporting a shared address space. There are five such nodes with 80 processors in total.

4.2.3 Parallel Algorithm on shared, distributed and hybrid architectures

In general, we assume N particles and P processors. We follow a master-worker model in a distributed architecture.

Distributed memory implementation: In a distributed computing environment, N/P particles are distributed among the processors. At each iteration, each processor calculates the local best position and communicates this information to the master processor through message passing primitives. The master processor computes the global best position from the local best positions received from all the processors. The master then broadcasts the global best position to all the processors, at which point, the processors start the next iteration. All particles terminate the algorithm after M iterations, where M is user defined. We use the MPI parallel programming language [1] for message passing.

Shared memory implementation: On a shared memory address space, each particle is considered as a thread.

[1] http://www.mcs.anl.gov/research/projects/mpi/

- A master thread creates N threads on a node, where N is the number of particles.
- Each thread (a particle) computes a new position and compares the newly computed position with local best position for update characteristics.
- Then each particle may update the local/global best value based on evaluation.
- The master thread computes and updates global best information.
- This continues until the algorithm reaches the maximum number of iterations.

We use the OpenMP [2] parallel programming language for implementation.

Hybrid model implementation: The parallel algorithm on a hybrid model assumes a hierarchical approach. A hybrid model is a multicore architecture with H nodes. Each of the node consists of C cores with P processors. The P processors use a shared memory address space. Communication between nodes is through message passing. We use OpenMP within the shared address space and MPI for communication between nodes. An MPI process runs on each node. Each process creates Q threads executed by P processors. There is one local master processor $master_L$ within each node and one central processor $master_G$. The threads (particle) use the same algorithm as described above in the shared memory model. The parallel algorithm for the hybrid model is given below.

- We distribute N/H particles on each node.
- Each process in a node creates Q threads. Assume Q=N/H.

[2] http://openmp.org/wp/

- The threads (particles) compute new position and compare the newly computed position with local best position for update characteristics.
- At the end of an iteration, each local master processor $master_L$ calculates the local global best value among the N/H particles and communicates this information to $master_G$, the central processor.
- The central processor computes the global best value and broadcasts it to all the other nodes, in particular $master_L$.
- This continues until termination.

Chapter 5

Experimental Results and Discussion

The experimental results for both sequential and parallel algorithms are presented here.

5.1 Sequential PSO Algorithm

Table 3.1 captures the PSO parametric conditions used in various works in the literature. We have used a combination of the parametric conditions in the sequential PSO based option pricing algorithm. The maximum number of iterations is set to 1000 and the number of particles N is defined as 20 for most of the simulations, and r is set to 0.05 for all simulations. We have considered both constant and dynamic volatility. We set the vectors $\overrightarrow{X_{max}}$ = [35, T] and $\overrightarrow{X_{min}}$ = [-35, 0].

First, the accuracy of the pricing results is tested against the Black-Scholes-Merton

model by varying volatility and contract period. Note that the Black-Scholes-Merton model assumes constant volatility. Therefore, to make a fair comparison between our PSO algorithm and the Black-Scholes-Merton model, we set the volatility statically at the start of the program and do not change during run time. The comparison results are shown in tables 5.1- 5.6.

The tables captures various constant volatilities and expiration time (T), the option exercise time (time) for the PSO, the PSO option value (PSO-value) and the Black-Scholes-Merton option value (BSM-value). Also these tables represent various initial conditions for the finance parameters such as S, K.

In these tables, we can see that our pricing results slightly deviate from Black-Scholes-Merton model. There are two major reasons for this: (i) the number of particles chosen (20) in these experiments is small. Therefore, there is a chance that the particles are unable to explore the entire search space; (ii) the number of iterations is 1000, which is also small.

To circumvent this problem we experimented with $N = 40$ and $10,000$ iterations. Table 5.7 presents the results with S=25, K=20 for 10000 iterations and 40 particles. It is easy to observe that the PSO option values computed with large number of iterations and particles are comparable with Black-Scholes-Merton model.

Table 5.8 presents the results with S=22 and K=26. It is easy to observe that the PSO option values presented in table 5.8 are comparable to the option values from the Black-Scholes-Merton model.

From this, we can conclude that if the number of iterations is increased together with number of particles, our PSO algorithm would converge to the Black-Scholes-

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.408	4.62	5.94
	0.4	0.40	6.68	6.14
	0.5	0.41	7.90	6.6
0.75	0.3	0.54	7.68	6.21
	0.4	0.55	7.45	6.72
	0.5	0.34	10.15	7.31
1	0.3	0.525	7.71	6.62
	0.4	0.66	9.67	7.24
	0.5	0.56	9.41	7.95

Table 5.1: Call option value for $S = 25$, $K = 20$

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.52	0.85	0.44
	0.4	0.54	1.94	0.89
	0.5	0.438	1.66	1.39
0.75	0.3	0.45	1.44	0.73
	0.4	0.58	1.84	1.41
	0.5	0.48	3.13	2.07
1	0.3	0.16	1.97	1.14
	0.4	0.38	2.36	1.89
	0.5	0.49	2.72	2.68

Table 5.2: Call option value for $S = 20$, $K = 25$

Merton exact solution for European option. Since this would increase computation time, we designed a parallel algorithm and the implementation results are shown in the next section.

Tables 5.9 and 5.10 present option values captured by our algorithm for random volatility. With these experiments we can conclude that PSO is a viable alternative approach for pricing options.

In the next experiment, we determine the execution time of the PSO algorithm.

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.25	5.8	5.84
	0.4	0.18	6.01	6.24
	0.5	0.12	6.25	6.72
0.75	0.3	0.55	6.13	6.3
	0.4	0.23	6.48	6.85
	0.5	0.25	6.6	7.48
1	0.3	0.66	6.57	6.73
	0.4	0.64	7.05	7.4
	0.5	0.29	7.54	8.15

Table 5.3: Call option value for S=26, K=21

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.21	0.68	0.5
	0.4	0.14	0.94	0.98
	0.5	0.13	1.2	1.52
0.75	0.3	0.4	1.05	0.88
	0.4	0.42	1.25	1.54
	0.5	0.47	1.81	2.24
1	0.3	0.65	1.52	1.25
	0.4	0.31	1.58	2.05
	0.5	0.49	2.55	2.88

Table 5.4: Call option value for S=21, K=26

For this, we use the standard sequential binomial lattice algorithm. Table 5.11 presents option values computed by binomial lattice and our sequential PSO-based algorithm; also presented in this table are the execution time for both methods. Note that the number of iterations of our PSO-based algorithm is equivalent to time steps of binomial lattice. The option value is computed for $S = 21$, $K = 25$, $\sigma = 0.3$, $N = 20$, $r = 0.05$.

Similarly, Figure 5.1 presents execution time of our sequential and binomial lattice

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.33	4.61	5.04
	0.4	0.35	4.84	5.53
	0.5	0.37	5.04	6.07
0.75	0.3	0.2	4.77	5.56
	0.4	0.61	5.34	6.19
	0.5	0.1	5.53	6.88
1	0.3	0.74	5.36	6.03
	0.4	0.73	5.8	6.78
	0.5	0.82	6.2	7.59

Table 5.5: Call option value for S=26, K=22

T	σ	**time**	**PSO-value**	**BSM-value**
0.5	0.3	0.12	0.7	0.75
	0.4	0.33	0.78	1.31
	0.5	0.37	1.41	1.9
0.75	0.3	0.48	0.9	1.2
	0.4	0.29	1.17	1.93
	0.5	0.48	1.44	2.49
1	0.3	0.79	1.22	1.63
	0.4	0.25	1.53	2.49
	0.5	0.44	1.88	3.37

Table 5.6: Call option value for S=22, K=26

model. We observe that binomial lattice works faster than our sequential algorithm. Note that in the PSO algorithm, we use random number generators for the parameters r_1 and r_2 in Equation 4.3 for each of the N particles at each iteration. During each iteration, to avoid duplication we discard some of the random numbers generated among particles and regenerate more random numbers. This adds to overhead and the cost of the overhead in PSO in presented in the table 5.11 as well. For future work, we plan to use parallel pseudo-number generator to reduce this overhead cost.

T	σ	**time**	**PSO-value**	**BSM-value**	**Itr**
0.5	0.5	0.34	6.85	6.6	10000
0.75	0.5	0.29	7.49	7.31	10000
1	0.4	0.63	7.99	7.24	10000

Table 5.7: Call option value for $S = 25$, $K = 20$, N=40

T	σ	**time**	**PSO-value**	**BSM-value**	**Itr**
1	0.4	0.27	1.62	2.49	1000
1	0.4	0.22	1.66	2.49	10000
1	0.5	0.52	1.97	3.37	1000
1	0.5	0.47	2.19	3.37	10000

Table 5.8: Call option value for $S = 22$, and $K = 26$, $N = 40$

T	σ	**value**	**time**
0.5	rand	3.9	0.3
0.75	rand	5.2	0.14
1	rand	6.2	0.8

Table 5.9: Call option value with random volatility for $S = 21$, $K = 23$

T	σ	**value**	**time**
0.5	rand	1.42	0.5
0.75	rand	2.17	0.67
1	rand	3.00	0.8

Table 5.10: Call option value with random volatility for $S = 21$, $K = 23$

Algorithm	**time step/iterations**	**value**	**Exec. time**	**PSO overhead**
Binomial	1000	5.90	0.0271	-
PSO-based	1000	5.1	0.1819	0.0688
Binomial	10000	5.90	2.4724	-
PSO-based	10000	5.4	6.71	3.1471

Table 5.11: Binomial lattice vs sequential PSO-based algorithm

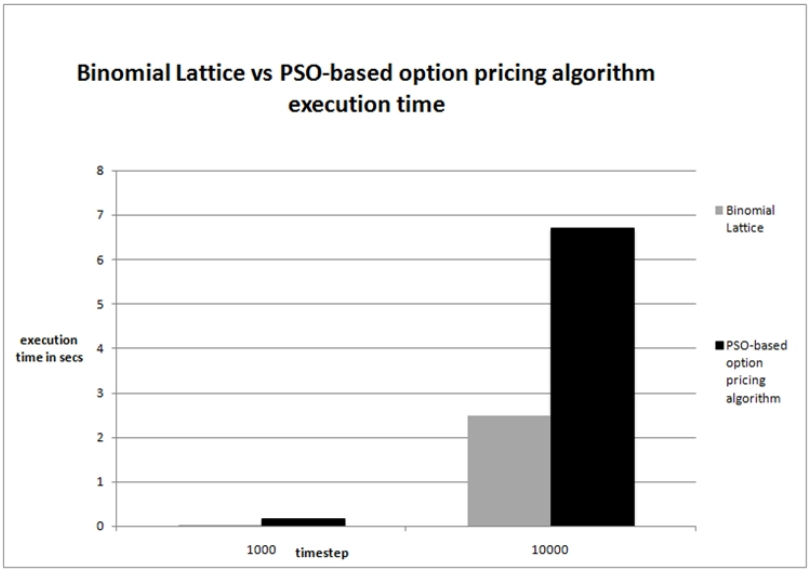

Figure 5.1: Binomial lattice vs sequential PSO-based algorithm

5.2 Parallel PSO Algorithm

We have implemented and tested our parallel algorithm on Helium cluster, a cluster of Linux servers available at the University of Manitoba, Department of Computer Science. The cluster consists of one head node and five computing nodes. The head node is a Sun Fire X4200 machine. Its configuration is as follows:

Processor: one Dual-Core AMD Opteron 252 2.6GHz

Memory: 2.0GB

Cache size : 1MB/each core

OS: CentOS 5

The five computing nodes are named as helium-01, helium-02, ..., helium-05. Each is a Sun Fire X4600 machine. Its configuration is as follows.

Processor: eight Dual-Core AMD Opteron 885 2.6GHz (totally 16 cores each node)

Memory: 32GB

Cache size : 1 MB L2/each core

OS: CentOS 5

CPU Interconnect: HyperTransport speed (1GHz, 8GB/s)

The parallel algorithm is implemented on a cluster of four nodes ($H = 4$) with eight dual-core processors, 64 homogeneous multi-core processors in total. We test

T	σ	**PSO-value**	**time**	**BSM-value**
0.5	0.3	4.47	0.16	4.96
0.5	0.4	4.71	0.32	5.42
0.5	0.5	5.01	0.22	5.94
0.75	0.3	4.69	0.20	5.45
0.75	0.4	5.03	0.35	6.05
0.75	0.5	5.21	0.19	6.71
1	0.3	5.01	0.15	5.9
1	0.4	5.87	0.92	6.61
1	0.5	5.94	0.19	7.38

Table 5.12: Parallel PSO-based call option value for $S = 25$ and $K = 21$

the algorithm on a shared memory address space, distributed message passing environment and hybrid (shared address space and distributed memory) environment.

First, we check the accuracy of our parallel algorithm results. In tables 5.12 and 5.13, we present option value computed using our parallel algorithm implemented on a shared memory address space (8 dual-core or 16 processors). We compare our pricing results with Black-Scholes-Merton model for different time period and volatility for two cases ($S = 25$, $K = 21$, $r = 0.05$, $N = 20$ and $Itr = 1000$) and ($S = 21$, $K = 25$, $r = 0.05$, $N = 20$ and $Itr = 1000$). The result shows that the option value computed from our parallel algorithm is comparable with Black-Scholes-Merton model in most of the cases.

We discuss Figure 5.2 observations in terms of speedup in subsection 5.2.1.

Figures 5.3, 5.4, and 5.5 show the execution time results with respect to number of particles for three different number of iterations, 1000, $10,000$ and $30,000$. These results show that as the number of iterations increases, the sequential PSO algorithm's performance degrades.

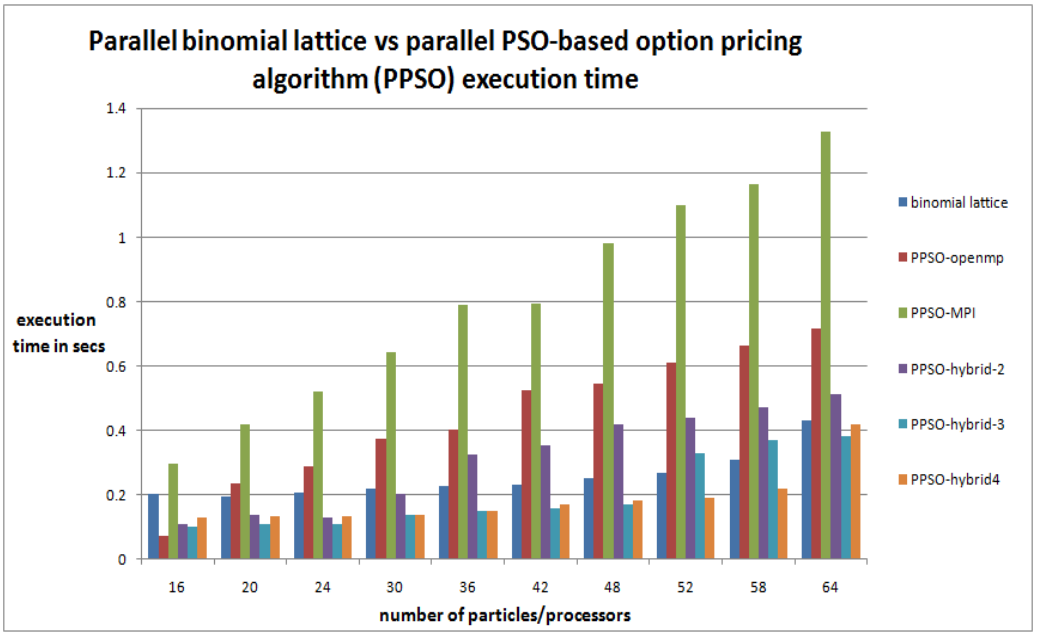

Figure 5.2: Parallel binomial vs parallel PSO-based algorithm

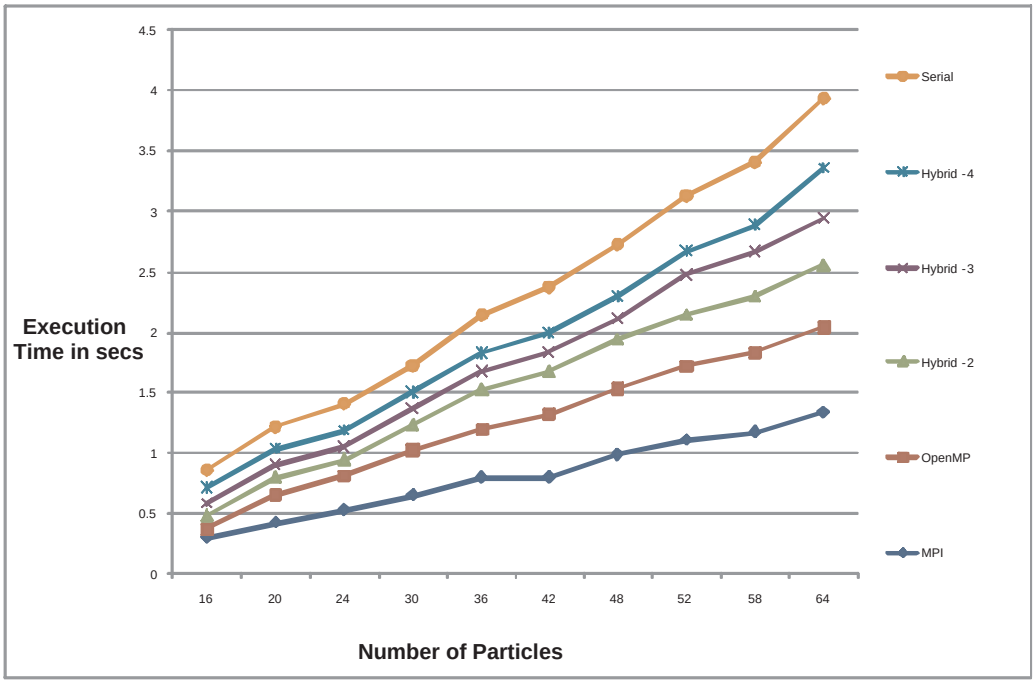

Figure 5.3: Execution Time for $Itr = 1000$

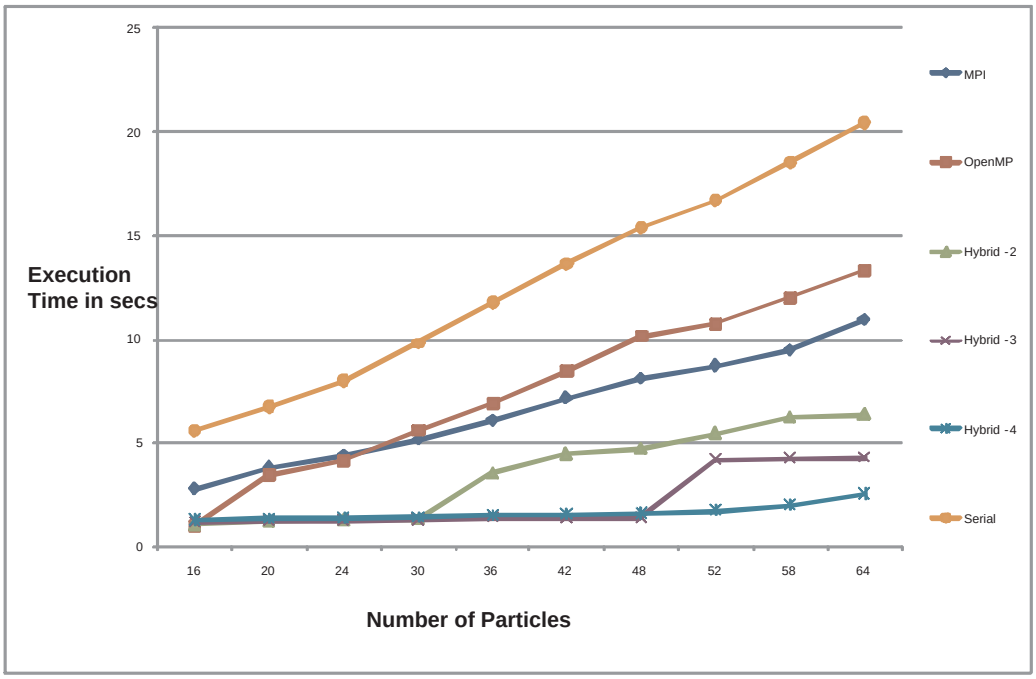

Figure 5.4: Execution Time for $Itr = 10000$

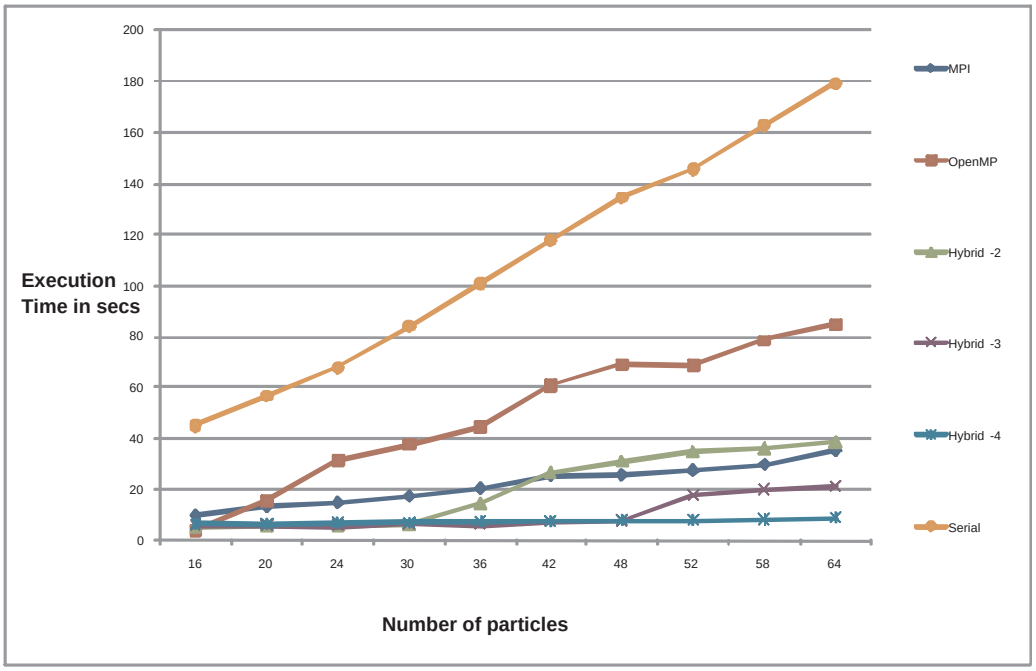

Figure 5.5: Execution Time for $Itr = 30000$

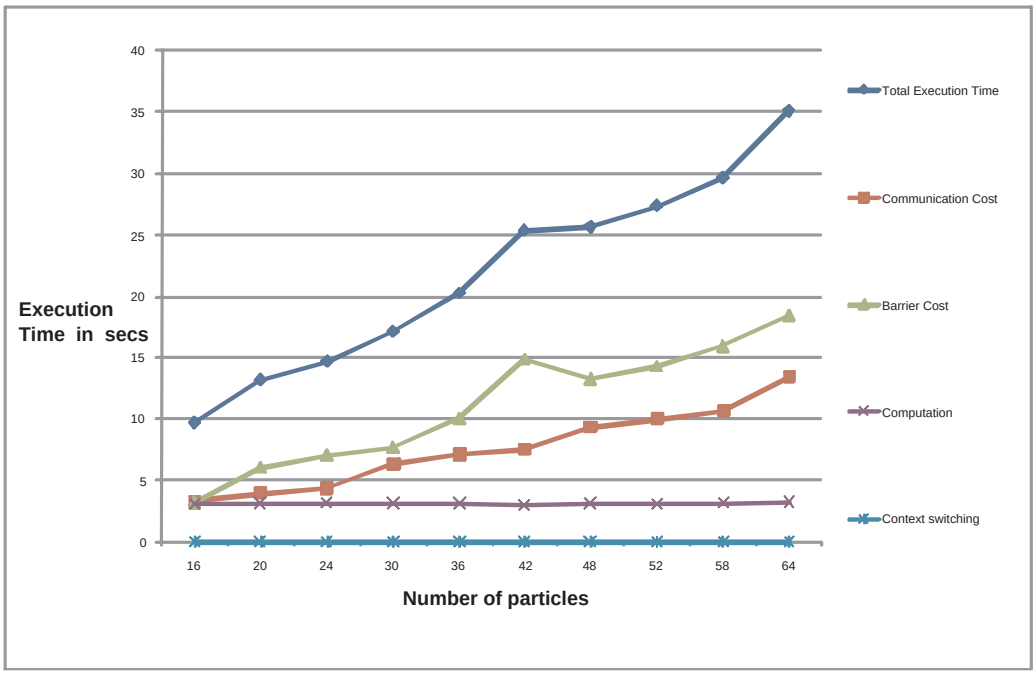

Figure 5.6: Execution Time Analysis in MPI for $Itr = 30000$

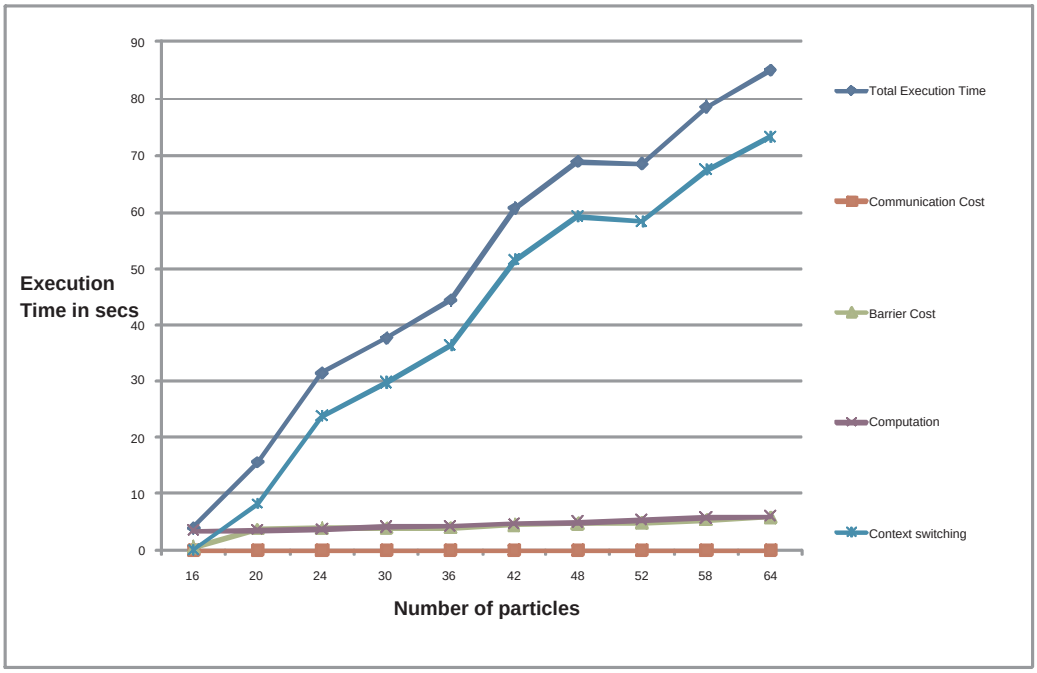

Figure 5.7: Execution Time Analysis in openMP for $Itr = 30000$

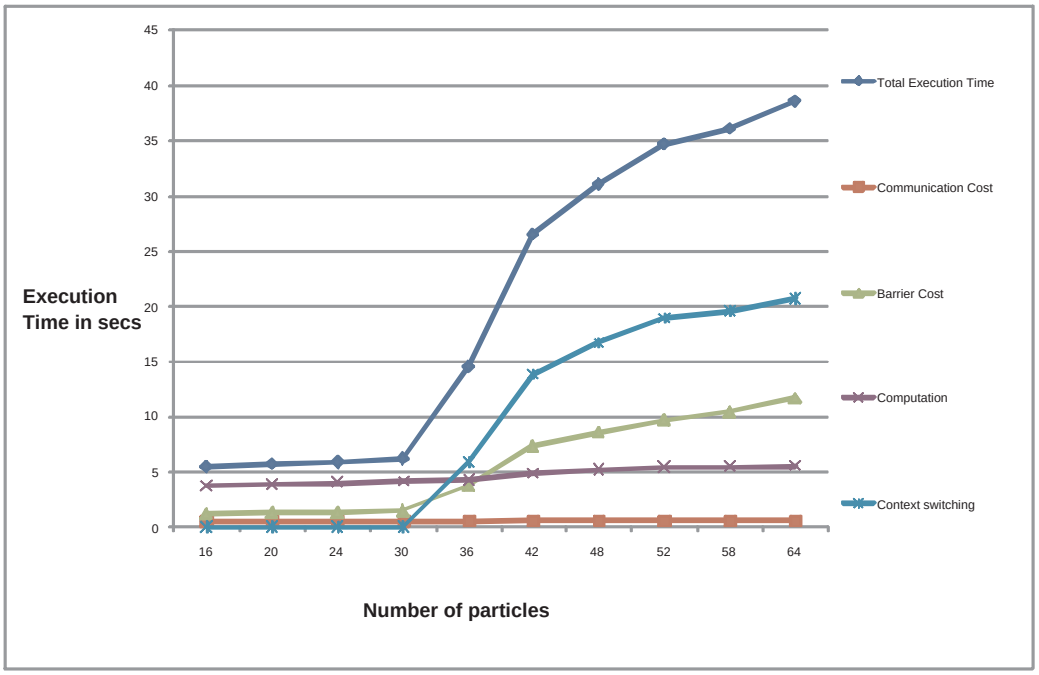

Figure 5.8: Execution Time Analysis in Hybrid-2 for $Itr = 30000$

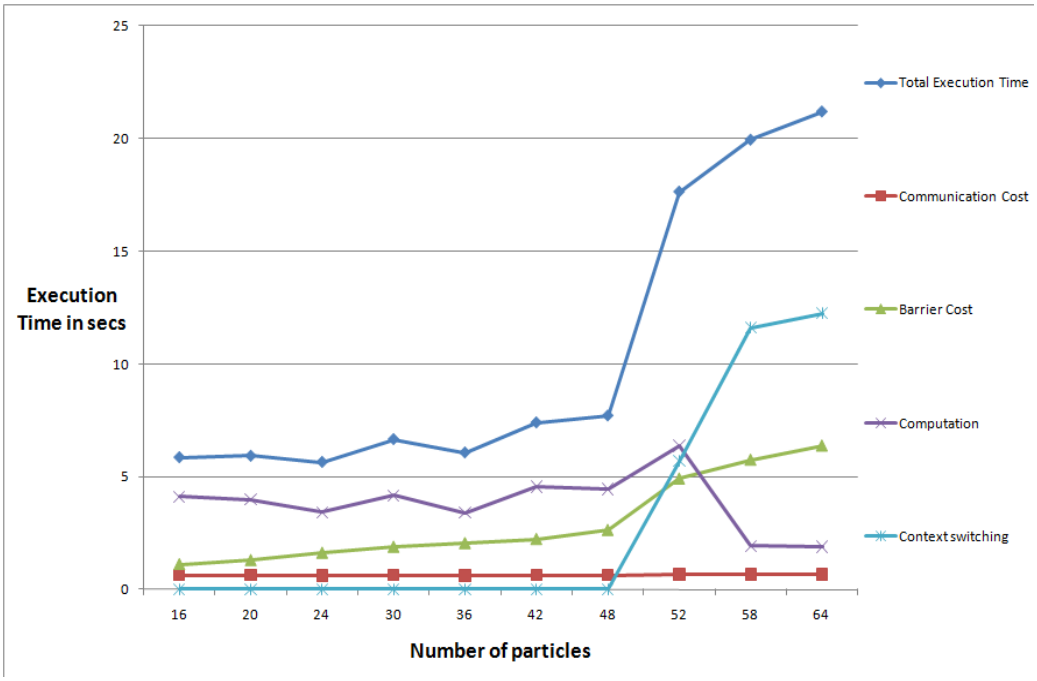

Figure 5.9: Execution Time Analysis in Hybrid-3 for $Itr = 30000$

T	σ	**PSO-value**	**time**	**BSM-value**
0.5	0.3	0.49	0.22	0.68
0.5	0.4	0.64	0.13	1.21
0.5	0.5	0.96	0.17	1.77
0.75	0.3	0.66	0.49	1.1
0.75	0.4	0.93	0.5	1.8
0.75	0.5	1.37	0.34	2.51
1	0.3	1.22	0.93	1.5
1	0.4	1.43	0.43	2.33
1	0.5	1.84	0.48	3.16

Table 5.13: Parallel PSO-based call option value for $S = 21$ and $K = 25$

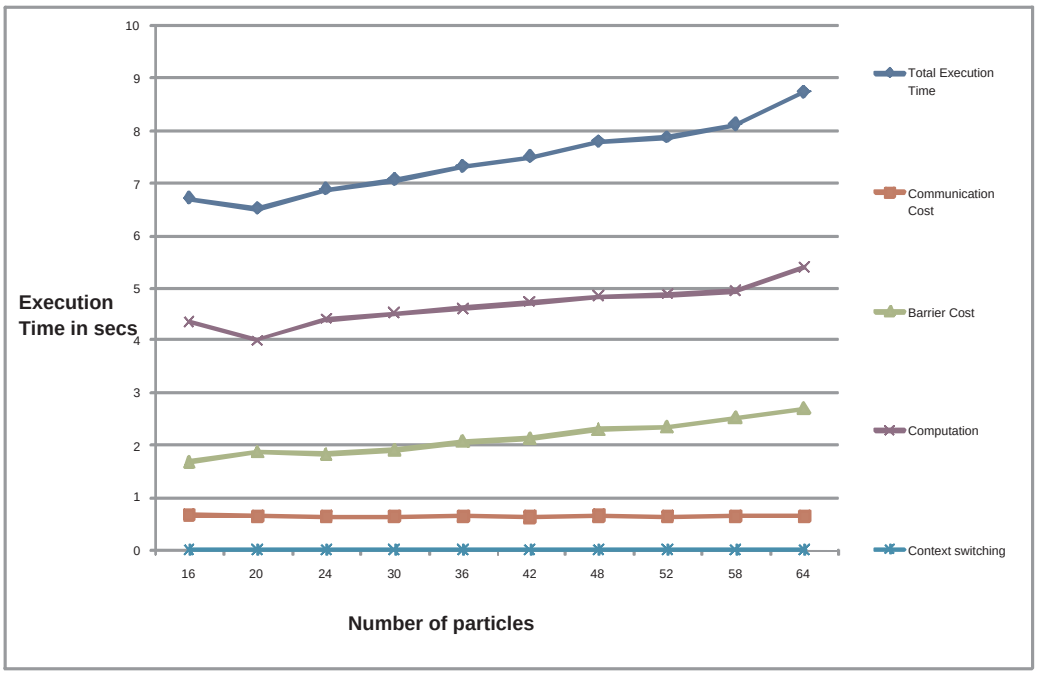

Figure 5.10: Execution Time Analysis in Hybrid-4 for $Itr = 30000$

Figures 5.6-5.10 and Tables 5.14-5.18 present detailed costs (computation, communication, barrier, and context switching) of our parallel algorithm on the three implementations for $30,000$ iterations. From these figures and tables, we can see

N	**Total**	**Communication**	**Barrier**	**Computation**	**Context Switching**
16	9.63706	3.34837	3.215363	3.073327	0
20	13.138794	3.952786	6.063423	3.122585	0
24	14.624433	4.398757	7.07041	3.155266	0
30	17.165423	6.339603	7.716524	3.109296	0
36	20.245002	7.134161	10.003356	3.107485	0
42	25.337552	7.534619	14.819025	2.983908	0
48	25.6659	9.341526	13.210293	3.114081	0
52	27.322814	9.977233	14.249422	3.096159	0
58	29.599099	10.611122	15.850164	3.137813	0
64	35.04949	13.366944	18.465184	3.217362	0

Table 5.14: MPI Execution Time (in secs) Analysis

N	**Total**	**Communication**	**Barrier**	**Computation**	**Context Switching**
16	3.916972	0	0.524821	3.40686	0
20	15.527018	0	3.782566	3.60123	8.143222
24	31.261782	0	3.85937	3.76315	23.639262
30	37.581378	0	3.891961	4.140686	29.548731
36	44.371843	0	3.950918	4.23616	36.184765
42	60.587454	0	4.448343	4.67812	51.460991
48	68.851265	0	4.690366	4.94731	59.213589
52	68.452747	0	4.82593	5.3686	58.258217
58	78.460243	0	5.41393	5.740612	67.305701
64	84.969199	0	5.826544	5.99621	73.146445

Table 5.15: OpenMP Execution Time (in secs) Analysis

that context switching and communication cost is reduced in hybrid implementation. Also, as number of particles increases, in each hybrid model (hybrid-2, hybrid-3, and hybrid-4) , the communication cost almost remain similar. However, as the number of particles increase the barrier cost increases. This is due to the synchronization barrier that is needed when the central processor computes the global result. To overcome the barrier costs, as future work, a parallel asynchronous PSO based option pricing

N	**Total**	**Communication**	**Barrier**	**Computation**	**Context Switching**
16	5.495242	0.523812	1.224079	3.747351	0
20	5.70468	0.523825	1.333155	3.8477	0
24	5.900409	0.524181	1.345766	4.030462	0
30	6.209401	0.538388	1.504567	4.166446	0
36	14.518815	0.560297	3.803194	4.291724	5.8636
42	26.534724	0.580799	7.341777	4.847948	13.7642
58	31.068775	0.597944	8.601333	5.218298	16.6512
52	34.701664	0.605067	9.720564	5.393723	18.98231
58	36.07178	0.594344	10.46071	5.424626	19.5921
64	38.567129	0.609588	11.686713	5.527228	20.7436

Table 5.16: Hybrid-2 Execution Time Analysis

N	**Total**	**Communication**	**Barrier**	**Computation**	**Context Switching**
16	5.8553	0.629113	1.10415	4.122037	0
20	5.9169	0.633984	1.285973	3.996943	0
24	5.6446	0.619135	1.610317	3.415148	0
30	6.6581	0.623361	1.876876	4.157863	0
36	6.0665	0.619495	2.03861	3.408395	0
42	7.3985	0.623361	2.22361	4.551529	0
48	7.7048	0.63327	2.62652	4.44501	0
52	17.6403	0.659233	4.912761	5.872006	7.1963
58	19.9456	0.658865	5.748794	7.320341	7.6176
64	21.1962	0.665801	6.382889	6.29561	9.2519

Table 5.17: Hybrid-3 Execution Time Analysis

algorithm can be explored.

5.2.1 Parallel Algorithms' Speedup

Table 5.19 shows the execution time for parallel binomial lattice algorithm and PSO algorithm on a distributed memory architecture. We notice that the PSO paral-

N	**Total**	**Communication**	**Barrier**	**Computation**	**Context Switching**
16	6.705214	0.669102	1.67314	4.362972	0
20	6.510182	0.643899	1.865976	4.000307	0
24	6.878896	0.640903	1.82014	4.417853	0
30	7.063141	0.635949	1.902493	4.524699	0
36	7.32036	0.643695	2.057069	4.619596	0
42	7.493296	0.629022	2.124872	4.739402	0
48	7.78891	0.653641	2.28721	4.848059	0
52	7.86331	0.636495	2.33722	4.889595	0
58	8.113185	0.643652	2.514872	4.954661	0
64	8.729292	0.643695	2.684872	5.400725	0

Table 5.18: Hybrid-4 Execution Time Analysis

N	**parallel binomial**	**parallel PSO-based algorithm**	**Itr**
16	0.202	0.2959	1000
20	0.194	0.4187	1000
24	0.205	0.5218	1000
30	0.221	0.6445	1000
36	0.229	0.7881	1000
42	0.233	0.7917	1000
48	0.252	0.9814	1000
52	0.269	1.0997	1000
58	0.307	1.1639	1000
64	0.431	1.3286	1000

Table 5.19: Parallel execution time in a distributed memory architecture

lel execution time exceeds that of binomial lattice method. This is mainly due to (i) overhead caused by PSO computations for particle position update, and (ii) random number generation (mentioned earlier). These points and the communication bottleneck on distributed memory machines motivated us to consider the shared address space and hybrid architectures.

In the tables 5.20-5.22, hybrid-H represents H nodes with 8 dual-cores or 16 pro-

N	**openmp**	**hybrid-2**	**hybrid-3**	**hybrid-4**	**Itr**
16	0.072	0.1079	0.1026	0.1292	1000
20	0.2338	0.1355	0.1093	0.132	1000
24	0.2871	0.1287	0.1102	0.1344	1000
30	0.3747	0.2034	0.139	0.1362	1000
36	0.4042	0.3265	0.15	0.1495	1000
42	0.5229	0.353	0.1583	0.1712	1000
48	0.5449	0.4167	0.1703	0.1836	1000
52	0.6121	0.4374	0.3289	0.1904	1000
58	0.662	0.4719	0.3682	0.218	1000
64	0.7169	0.513	0.3818	0.417	1000

Table 5.20: Parallel execution time in shared memory and hybrid architecture

cessors per node. Table 5.20 shows the execution time on shared memory and hybrid architectures. The parallel PSO algorithm on the hybrid architecture performs better than the shared memory architecture as we increase the number of particles. In shared memory architecture, due to context switching the execution time is higher as we increase the number of particles.

Tables 5.21 and 5.22 present speedup results of parallel PSO-based option pricing algorithm on shared and hybrid architectures for larger number of iterations. The speedup [Grama et al., 2003] is defined as $\frac{T_1}{T_P}$ where T_1 is the execution time of the sequential PSO algorithm on one processor and T_P is the execution time for parallel PSO algorithm on P processors.

First, let us consider the shared memory results. We observe that for 16 particles the speedup for 30000 iterations is higher than that of 10000 iterations. As the number of particles increases, in both tables we notice that the shared memory speedup results is significantly lower. The reason for this is as follows. The implementation of the

PSO algorithm is performed on 16 processors. In the case of $N = 16$, each particle, regarded as a thread, is allocated to one processor. When $N > 16$, say $N = 36$, each processor is allocated two particles with the exception of four processors allocated with three particles. In this case, the work is not well balanced. Also, in shared memory machines, the processors context switch between threads, which creates additional overhead.

Next, we compare the hybrid implementation results. Recall that hybrid-H represents H nodes with P processors. In our cluster, there are 16 processors per node. Hybrid-2 refers to 2 nodes with 32 processors, hybrid-3 refers to 3 nodes with 48 processors and hybrid-4 refers to 4 nodes with 64 processors. Also, recall that there is shared memory within each node and nodes communicate through message passing. Initially, we distribute N/H particles among the H nodes.

Among all the three hybrid cases, hybrid-4 produces better speedup results for both $10,000$ and $30,000$ iterations. We explain this as follows. When $N = 64$, each particle is allocated to each of the 64 processors in hybrid-4 cluster. Since shared memory performs better if there is one to one mapping of processors to threads as explained above for the shared memory results, the results for N = P produces the desired speedup results for hybrid-4 cluster. When $N < 64$, some of the processors in hybrid-4 cluster are idle. For example, when $N = 48$, each node is allocated $48/4 = 12$ particles per node. In this case, four processors per node are idle. When $N = 16$, $16/4 = 4$ particles are allocated per node, with 12 processors idle per node. As we increase N, we are able to reduce number of idle processors through load balancing. Therefore, we notice increase in the speedup.

We can extend this observation to the other hybrid-H cases. When N = 48, hybrid-3 performs well. Again, this is due to the one to one mapping between particles and processors. In hybrid-3, unlike hybrid-4, some processors may be idle or may have more particles per processor. For example, in the case when $N = 36$, $36/3 = 12$ particles are allocated to each node. In this case, four processors/node are idle. As explained in the hybrid-4 case, this load imbalance creates poor speedup. We can see from the tables that for all values of $N < 48$, the speedup with hybrid-3 is not comparable to N = 48. Now, when $N = 64$ ($N > 48$), $64/3 = 21$ particles are allocated per node. Each particle is scheduled to each of the 16 processors/node with five processors in each node allocated one extra particle. As mentioned in the shared memory results there is more work for some processors (unbalanced workload) and therefore the speedup results fall below that of $N = 48$.

Note that, in the parallel PSO algorithm for the hybrid case, particles within a node compute independently of the other particles in other nodes until the calculation of global best. The master processor for each node, $master_L$, gathers the global local best results from each of its assigned particles, and sends the result to the central processor, $master_G$. At the end of each iteration, a comparison among the local best values (determined by $master_L$ in each node) are done by $master_G$. Since the computations within a node are local, the amount of communication to the master, $master_G$ is very low. That is, each node communicates its local best among its assigned particles and $master_L$ sends one best value to the $master_G$. In general, for hybrid-H, there is basically 2*(H-1) amount of communication between the nodes and $master_G$. Therefore, for I iterations, 2*(H-1)*I communication is performed. This

N	**openmp**	**hybrid-2**	**hybrid-3**	**hybrid-4**	**Itr**
16	5.63	5.13	4.73	4.42	10000
20	1.95	5.45	5.45	4.94	10000
24	1.93	6.33	6.49	5.75	10000
30	1.75	7.44	7.75	6.98	10000
36	1.71	3.33	8.43	7.78	10000
42	1.62	3.06	10.08	8.80	10000
48	1.51	3.28	11.25	9.66	10000
52	1.55	3.07	4.01	9.68	10000
58	1.54	2.98	4.38	9.28	10000
64	1.53	3.22	4.77	8.09	10000

Table 5.21: Parallel PSO-based algorithm speedup in shared and hybrid architecture

does not seem to cause a performance degradation issue as can be seen from N = P cases in each of the hybrid-H results.

We conclude from these results that, for a shared memory architecture or a hybrid architecture, a one to one mapping of particles to processors is recommended for performance speedup. Load imbalance and idle processors degrade the performance. A hybrid architecture reduces the communication overhead cost that is incurred in a pure distributed memory architecture. It is also best if we reduce the communication as much as possible.

N	**openmp**	**hybrid-2**	**hybrid-3**	**hybrid-4**	**Itr**
16	11.47	8.17	7.67	6.70	30000
20	3.63	9.87	9.52	8.65	30000
24	2.16	11.44	11.96	9.82	30000
30	2.24	13.53	12.62	11.89	30000
36	2.27	6.95	16.63	13.78	30000
42	1.94	4.43	15.90	15.70	30000
48	1.95	4.32	17.44	17.25	30000
52	2.12	4.19	8.25	18.50	30000
58	2.07	4.51	8.15	20.05	30000
64	2.11	4.64	8.45	20.51	30000

Table 5.22: Parallel PSO-based algorithm speedup in shared and hybrid architecture

Chapter 6

Conclusion and Future work

The major contribution from this work is the conceptualization of PSO for the option pricing problem, and design of sequential and parallel algorithms from fundamental principles of PSO. To the best of our knowledge, this is the first attempt to use Particle Swarm Optimization for an American option pricing problem.

We studied both velocity equations (Shi and Eberhart (Equation 3.6) and Clerc and Kennedy (Equation 3.8)) to incorporate volatilities for option pricing, and chose Shi and Eberhart velocity equation (Equation 3.6) to implement. In both velocity equations, we consider constant and dynamic volatility. The random part of ω acts as volatility of the underlying asset in Shi and Eberhart velocity equation.

The particles collaborate in achieving the objective of exercising a given option at the best possible time that gives optimal option price results. The collaboration is achieved through local best (for a particle) and global best (for swarm) concepts. The discounting feature for an option is carefully employed for the local and global best computation to achieve the objective.

We have studied the performance evaluation of PSO-based sequential and parallel option pricing algorithms. First, we tested the accuracy of the algorithm against Black-Scholes-Merton model for various volatility and contract period. The PSO algorithm converges to Black-Scholes-Merton model when the number of iterations increases together with the number of particles. The execution time of the sequential PSO algorithm is slightly higher in comparison to the binomial lattice algorithm. This we attribute to the overall cost of generating many random numbers by the particles at each iteration.

Since it is very important and critical to lock-in the profit making opportunities in the real market, we have also designed and developed a parallel algorithm to expedite the computing process. The parallel PSO-based algorithm is tested on three different architectures: distributed, shared, and hybrid using up to 64 processors. We compare speedup of our parallel algorithm against parallel binomial lattice. The speedup results show that the parallel PSO-based algorithm has better speedup than parallel binomial lattice algorithm. We conclude that for a shared memory architecture or a hybrid architecture, as we increase the problem size for better accuracy, it is advisable to use more number of processors in order to maintain a level of speedup. In other words, one-to-one mapping of particles to processors is desirable. We also noticed that hybrid architecture helps to reduce the communication overhead cost that incurs in distributed architecture. The speedup of our algorithm increases for larger number of iterations. The speedup is 20 in hybrid architecture using four computing nodes.

We observe that load imbalance and idle processors degrade the performance; as expected. To circumvent this problem, an asynchronous parallel algorithm can be

designed. Further improvement in our algorithm is possible by dynamically changing the PSO parametric conditions, finance parameters, and termination condition for the algorithm, which we leave as future work. Along with these, the PSO-based option pricing algorithms can be also extend for complex options.

Bibliography

S. Barua, R. K.Thulasiram, and P. Thulasiraman. High performance computing for a financial application using fast Fourier transform. In *Proceedings of the European Parallel Computing Conference, (EuroPar 2005)*, volume 3648, pages 1246–1253, Lisbon, Portugal, 2005.

M. A. J. Bharadia, N. Christofides, and G. R. Salkin. Computing the Black-Scholes implied volatility: Generalization of a simple formula. *Advances in Futures and Options Research*, 8:15–29, 1995.

M. A. J. Bharadia, N. Christofides, and G. R. Salkin. A quadratic method for the calculation of implied volatility using the Garman-Kohlhagen model. *Financial Analysts Journal*, 52:61–64, 1996.

B. Birge. PSOt-a particle swarm optimization toolbox for use with Matlab. In *Proceedings of the 2003 IEEE swarm intelligence symposium*, pages 182–186, December 2003.

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

P. Boyle. Options: A Monte Carlo approach. *Journal of Financial Economics*, 4: 223–238, 1977.

A. Brabazon and M. O'Neil. *Biologically Inspired Algorithms for Financial Modelling (Natural Computing Series)*. Springer-Verlag, 2006.

B. Bullnheimer, R. F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. In *Annals of Operations Research*, volume 89, pages 319–328, 1999.

G. D. Caro, F. Ducatelle, and L. M. Gambardella. AntHocNet: An adaptive nature inspired algorithm for routing in mobile ad hoc networks. *European Transactions on Telecommunications (Special Issue on Self-Organization in Mobile Networking)*, 16:443–455, 2005.

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

P. Chalasani, S. Jha, F. Egriboyun, and A. Varikooty. A refined binomial lattice for pricing American Asian options. *Review of Derivatives Research*, 3:85–105, January 1999.

D. M. Chance. A generalized simple formula to compute the implied volatility. *The Financial Review*, 3:859–867, 1996.

S.-H. Chen, W.-C. Lee, and C.-H. Yeh. Hedging derivative securities with genetic programming. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 8:237–251, 1999.

Y. Chen, B. Yang, and J. Dong. Evolving flexible neural networks using ant programming and PSO algorithm. *LNCS 3173*, pages 211–216, 2004.

Y. Chen, B. Yang, and J. Dong. Time series prediction using a local linear wavelet neural network. *Neurocomputing*, pages 449–465, 2005.

Y. Chen, B. Yang, and J. Dong. Time-series prediction using a local linear wavelet neural network. *Neurocomputing*, 69:449–465, 2006.

N. K. Chidambaran, C. W. J. Lee, and J. R. Trigueros. Option pricing via genetic programming. In *Computational Finance-Proceedings of the Sixth International Conference, Leonard N. Stem School of Business*, pages 583–598, January 1999.

M. Clerc and J. Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6:62–73, February 2002.

J. C. Cox, S. A. Ross, and M. Rubinstein. Options pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

M. Dorigo, V. Maniezzo, and A. Colorni. Ant system–optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics*, 26: 29–41, February 1996.

R. Eberhart and Y. Shi. Tracking and optimizing dynamic systems with particle swarms. In *Proceedings IEEE World Congresss on Evolutionary Computation*, pages 94–97, Seoul, Korea, 2001.

A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson Education Limited, 2003. ISBN 02016485652.

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

J. Hull. *Options, Futures, and other Derivative Securities*. Prentice Hall, 7 edition, May 2008.

J. M. Hutchinson, A. W. Lo, and T. Poggio. A nonparametric approach to pricing and hedging derivative securities via learning networks. *Journal of Finance*, 49: 851–889, 1994.

G. K. Jha, S. Kumar, H. Prasain, P. Thulasiraman, and R. K. Thulasiram. Option pricing using particle swarm optimization. In *Proceedings of the 2009 C* conference on computer science and software engineering (C3S2E) conference*, pages 267–272, May 2009a.

G. K. Jha, P. Thulasiraman, and R. K. Thulasiram. PSO based neural network for time series forecasting. In *Proceedings of IEEE International Joint Conference on Neural Networks*, pages 1422–1427, June 2009b.

C. Keber. Genetically derived approximations for determining the implied volatility. *OR Specktrum*, 21:205–238, 1999.

C. Keber and M. G. Schuster. Generalized ant programming in option pricing: determining implied volatilities based on American put options. In *IEEE Proceedings of Computational Intelligence in Financial Engineering*, pages 123–130, December 2003.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1958, March 1995.

B. Koh-II, A. D. George, R. T. Haftka, and B. J. Fregly. Parallel asynchronous particle swarm optimization. *International Journal For Numerical Methods in Engineering*, 67:578–595, January 2006.

S. Kumar, R. K. Thulasiram, and P. Thulasiraman. A bioinspired algorithm to price options. In *ACM Proc. of the C* Conference on Computer Science and Software Engienering*, pages 11–22, Montreal, May 2008.

S. Kumar, R. K. Thulasiram, and P. Thulasiraman. *ACO for Option Pricing*, volume 2 of *SCI 185*, chapter 4, pages 51–73. Springer-Verlag, 2009.

S. Lee, J. Lee, D. Shim, and M. Jeon. Binary particle swarm optimization for balck-scholes option pricing. *LNCS*, 4692:85–92, 2007.

J. Li, D. Wang, S. Chi, and X. Hu. An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration. *International Journal of Innovation Comupting, Information and Control*, 3(B(6)):1707–1714, December 2007.

V. Maniezzo and A. Colorni. The ant system applied to the quadratic assignment problem. *Knowledge and Data Engineering*, 11(5):769–778, 1999.

R. C. Merton. Theory of rational option pricing. *Bell Journal of Economics and Management Science*, 4:141–183, 1973.

J. Nenortaitė. A particle swarm optimization approach in the construction of decision-making model. *Information Technology and Control*, 36:449–465, 2007.

S. Rahmayil, I. Shiller, and R. K. Thulasiram. Different estimators of the underlying asset's volatility and option pricing errors: parallel Monte-Carlo simulation. In *Proceedings of the International Conference on Computational Finance and its Applications (ICCFA)*, pages 121–131, Bologna, Italy, 2004.

J. Schutte, B.J.Fregly, R. Haftka, and A.D.George. A parallel particle swarm optimizer. *International Journal For Numerical Methods in Engineering*, 61:2296–2315, October 2004.

Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proceedings IEEE World Congresss on Evolutionary Computation*, pages 69–73, Archorage, AK, USA, May 1998.

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

R. K. Thulasiram, L. Litov, H. Nojumi, C. Downing, and G. 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, 2001.

I. C. Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85:317–325, March 2003.

G. Venter and J. Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *Journal of Aerospace Computing, Information, and Communication*, 3(3):123–137, March 2006.

J. Wang, E. Osagie, P. Thulasiraman, and R. K.Thulasiram. Hopnet: A hybrid ant colony optimization routing algorithm for mobile ad hoc network. *Ad Hoc Networks*, 7:690–705, 2009.

H. F. Wedde, M. Farooq, T. Pannenbaecker, B. Vogel, C. Mueller, J. Meth, and R. Jeruschkat. BeeAdHOC: An energy efficient routing algorithm for mobile ad hoc networks inspired by bee behavior. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 153–160, Washington, DC, June 2005.

Z. Yin, A. Brabazon, and C. O'Sullivan. Adaptive genetic programming for option pricing. In *Proceedings of the 2007 conference companion on Genetic and evolutionary computation (GECCO)*, pages 2588–2594, London, England, UK, 2007.