An Implementation of Parallel Processing in Pricing and Hedging High-Dimensional American Options

by

Matthew Hall

A research project
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I am the sole author of this project. This is a true copy of the project, including any required final revisions, as accepted by my supervisors and readers.

I understand that my project may be made electronically available to the public.

Statement of Contributions

The following supervisors provided contributions to the project. The extent of their contributions is limited to overseeing and providing guidance on possible areas of research, access to research materials, how to tackle specific problems, and how to best present the outcomes of this project.

Supervisor(s): Hans De Sterck, Professor
Dept. of Applied Mathematics, University of Waterloo
Jun Liu, Associate Professor
Dept. of Applied Mathematics, University of Waterloo

Abstract

We present a parallel implementation of an existing method for computing prices and deltas of high dimensional American Options as a method of overcoming some of the known issues of computational time cost associated with the known method. The known method utilizes a sequence of Neural networks which learn the difference of the option pricing functions between timesteps by using a least squares residual derived from a backwards stochastic differential equation. Our method reduces the computational complexity of the known algorithm by the square of a known factor, at the cost of a limited reduction in accuracy. Our method also continues to compute prices and deltas across the same space-time as what is presented in the previous literature and is thus usable in the construction of delta-hedging processes for market traders. The numerical simulations of the method show that we reach significant reductions in the wall clock time required to complete the algorithm while still outputting accurate results within a certain statistical significance.

Acknowledgements

I would like to thank my supervisors for their extensive help along the way towards the creation and completion of this. From providing me with an assortment of research materials to helping direct me towards solving certain milestone issues that arose during the project, without Hans and Jun's help the completion of this project would have been significantly more difficult.

Dedication

This is dedicated to my parents, for supporting my education and extracurricular affairs for as long as I can remember.

Table of Contents

Chapter 1

Introduction

Option pricing is one of the oldest problems in the field of financial mathematics and has been the focus of many published works as mathematical tools have developed over time. Not only are market investors interested in calculating the fair price of an asset, but they also require the solutions of the first derivatives of the option values, known as the deltas (Hull [14]), in order to hedge their portfolios and minimize their exposure to risk. The level of risk-aversion in market participants drives the need for accurate delta hedging processes which are difficult to solve theoretically for any reasonably-sized dimension of options. European options are generally considered to be 'vanilla' types of options due to their less complex form and as a result are much easier to price, so plenty of works have targeted more complex types of options including Asian and American style option pricing and hedging problems. In terms of American options, which are the focus of this paper, past solution algorithms have ranged from Binomial trees [14], iterative solutions of free boundary partial differential equations (PDEs, Achdou and Pirroneau [1], Duffy [5], Forsyth and Vetzal [9], and Reisinger and Witte [19]), regression methods (Longstaff and Schwartz [18], Kohler [16], and Tsitsiklis and Van Roy [22]) as well as many others. These methods served researchers well in practical application, however more often than not these methods would suffer from what came to be known as 'the curse of dimensionality.' This meant that the computational requirements to obtain solutions to the problem increases rapidly with the growing size of the dimensionality of the underlying option. The curse of dimensionality is not exclusive however to the field of financial mathematics and many solutions have presented themselves depending on the application, however one of the most promising is the emergence of neural networks (Goodfellow *et al.* [12]) in the late 20^{th} and early 21^{st} century.

Previous works have utilized neural networks to price European options (E. *et al.* [6],

Beck *et al.* [3], Han *et al.* [13]) which is generally easier to compute than more complex options. Separate works have also used neural networks in American option pricing problems such as Chen and Wan [4], Fujii *et al.* [10], and Sirignano and Spiliopoulos [20], however these works are limited in certain aspects. Fujii *et al.* only solves the option pricing problem at a single domain point, Sirignano and Spiliopoulos' method only solves for American option prices on the space-time domain, but not deltas. Finally, Chen and Wan solves these issues by providing an algorithm which provides both option prices and deltas on the domain, but suffers from computational cost issues.

We present a method of implementing parallel processing to the algorithm of Chen and Wan by considering a multilevel grid on the domain to construct a more computationally efficient form while maintaining the same order of accuracy of the overall solution. We also provide theoretical analysis for expected memory costs and computational wall clock times of the methods proposed by Chen and Wan in [4], as well as our own proposed parallel method. Finally, we numerically explore the effects of the size of each level of the multilevel grid on the computational wall clock time used to complete the proposed parallel algorithm.

The paper is organized as follows: Chapter 2 formulates the option pricing and delta hedging problem that needs to be solved and discusses a general form of a solution. Chapter 3 discusses a previous algorithm devised by Chen and Wan [4] which the proposed parallel algorithm is based off of. Chapter 4 formulates the mathematical basis of the partition of the domain and the proposed parallel algorithm. Chapter 5 analyzes the theoretical computational cost saves of the parallel algorithm over previous solutions. Finally, Chapter 6 explores the numerical results of our simulations which provide evidence for our theoretical analysis. Chapter 7 provides a short conclusion to the paper.

Chapter 2

Problem Formulation

This chapter defines the underlying mathematical theory of the value of American options and the dynamic problem of pricing the option backwards in time by utilizing the backward stochastic differential equation (BSDE).

2.1 Financial Options

An option is a financial instrument whose value is calculated on the price of a set of one or more underlying assets. The buyer of the option, sometimes called the holder, is afforded the right to buy or sell the underlying assets at a fixed price K, known as the strike price, at some future time. The seller/writer of the option is obligated to fulfill the financial contract and sell or buy the single/basket of assets to the holder as stipulated by the form of the financial option. There are many parameters that define the kind of option such as when the option can be exercised, and the function which describes the amount of value the holder receives when utilizing their right to exercise the contract, known as the 'exercise value' of the option. The exercise value of any option can be written in the general form

$$f(\vec{s}) = \max(g(\vec{s}), 0), \tag{2.1}$$

where $\vec{s} = (s_1, s_2, \cdots, s_d)$ is the price vector for d assets. Known as the 'payoff function' of the option, the function is non-negative because if exercising the option would lead to a loss in value, the holder can simply choose to not exercise the option at all and let it expire if necessary. The function g can be a multitude of different functions depending on the context of the option, in most cases it's based on the current prices of the underlying

assets, but in some cases (lookback/Asian options), it can also be based on passed values of the assets, also known as the price path. In this paper we focus specifically on American options and the dynamic problem of pricing these options and calculating their deltas.

2.2 American Options

American options are unique from other options as the holder can choose to exercise the contract at any time between the selling date and the maturity date. At each point in time, the holder of the option must decide whether it is more profitable to exercise the option at the current time, or hold the option and either hope it is more profitable or risk it losing value in the future. To this end, there are two important values that must be known to determine the value of the option at any point in time. The first is the exercise value defined above in the previous section. For American options this can take many forms but more commonly we see the geometric call option which has a payoff function of the form $f(\vec{s}) = \max\left(\left(\prod_{i=1}^{d} s_i\right)^{\frac{1}{d}} - X, 0\right)$, where X is a previously agreed upon value of the basket of underlying options, known as the strike price.

The second value that must be known in order to find the value of the option is what is called the 'continuation value' of the option, which is the maximum discounted expected payoff value should the option not be exercised at the current time t and asset price vector $\vec{s}$. The continuation value is mathematically described by the following equation:

$$c(\vec{s}, t) = \max_{\tau \in [t,T]} \mathbb{E}\left[e^{-r(\tau - t)} f(\vec{S}(\tau)) | \vec{S}(t) = \vec{s}\right]. \tag{2.2}$$

where T is the maturity, or end date, of the option. At this time the option either must be exercised or else it expires worthless.

Given these two values, the value of the option v is simply the maximum of the two, which in turn defines whether it is financially optimal to exercise the option at the current time, or hold the option for some greater future expected value:

$$v(\vec{s}, t) = \max\left[c(\vec{s}, t), f(\vec{s})\right] = \begin{cases} c(\vec{s}, t), & c(\vec{s}, t) > f(\vec{s}), \\ f(\vec{s}), & O.W. \end{cases} \tag{2.3}$$

The underlying assets can be defined by the random price vector $\vec{S} = (S_1, ..., S_d)^T \in \mathbb{R}^d$ where given an initial price vector $\vec{s}^{\,0} \in \mathbb{R}^d$, the elements follow the underlying stochastic differential equations (SDEs):

$$dS_i(t) = (r - \delta_i)S_i(t)dt + \sigma_i S_i(t)dW_i(t), i = 1, \cdots, d, \tag{2.4}$$
$$\vec{S}(0) = \vec{s}\,^0. \tag{2.5}$$

Here r is the risk-free interest rate of the market, $\vec{\delta} = (\delta_1, ..., \delta_d)^T \in \mathbb{R}^d$ and $\vec{\sigma} = (\sigma_1, ..., \sigma_d)^T \in \mathbb{R}^d$ are the dividend rates and volatilities of each of the underlying assets, respectively, and $\vec{W} = (W_1, ..., W_d)^T \in \mathbb{R}^d$ defines a d-dimensional correlated Weiner process.

Finally we are not only interested in the option value at any point in time, but also the option deltas, which are the first derivatives of the option value with respect to the asset prices, $\vec{\nabla} v(\vec{s}, t) \equiv \left(\frac{\partial v}{\partial s_1}(\vec{s}, t), \cdots, \frac{\partial v}{\partial s_d}(\vec{s}, t)\right)^T$. There are many other derivatives that are interesting in terms of financial options, sometimes called the 'greeks' by financial theorists (deltas, gammas, etc.), however the scope of this paper sticks to only calculating the deltas. The greeks are essential to market traders as they can be utilized to construct portfolios which reduce or eliminate exposure to financial risk, also known as 'hedging' the option. In continuous time, a perfect delta-hedging portfolio can completely eliminate the risk of holding the option.

2.3 Backward Stochastic Differential Equation (BSDE)

In Chen and Wan [4], the authors utilize the following theorem to convert the problem into a backward stochastic differential equation problem.

Theorem 2.1 (BSDE formulation). Assume that an American option is not exercised in the timespace $(t, t + dt)$. Then the continuation price of an American option at time t satisfies the following BSDE:

$$dc(\vec{S}, t) = rc(\vec{S}, t)dt + \sum_{i=1}^{d} \sigma_i S_i(t) \frac{\partial c}{\partial s_i}(\vec{S}, t)dW_i(t), \tag{2.6}$$

where $\vec{S}$ satisfies the SDE (2.4) and r, σ_i, and $dW_i(t)$ are the same as in (2.4).

Proof. The intricacies of the stochastic calculus which leads to this result are beyond the scope of this paper but interested readers are directed to the proofs in El Karoui et al. (1997) and Leentvaar (2008) which prove it through the use of Ito's lemma. □

The solution of the BSDE not only yields the option value, but subsequently also outputs the deltas, allowing the solution of both values of interest concurrently which is pivotal to the construction of a complete hedging process across both the spatial and temporal domain.

2.3.1 Discretization of the BSDE and Least Squares Solution

In the original paper by Chen and Wan [4] they utilize a Monte Carlo method with Euler time stepping in order to simulate the propagation of the asset values from the initial value according to SDE (2.4-2.5). Let $m = 1, \cdots, M$ index the individual price paths (simulations) and let $n = 0, \cdots, N$ index the discrete time points to evaluate the simulation between $t_0 = 0$ to $t_N = T$, we take time step size $\Delta t = \frac{T}{N}$. The simulation of the asset prices requires the ability to construct a realization of a d-dimensional correlated Weiner process. To do so we let $\rho \in \mathbb{R}^{d \times d}$ define the correlation matrix between the d assets, and we can decompose the matrix into its Cholesky factors such that $\rho = LL^T$. Given then a set of d realizations of a standard normal random variables $\Phi_i(t) \sim N(0,1)$, $i = 0, \cdots, d$, we can construct the realization of a d-dimensional correlated Weiner process over a single time step under the following equation:

$$(\Delta W_i)_m^n = \sum_{j=1}^{d} L_{ij} \, (\Phi_j)_m^n \, \sqrt{\Delta t}. \tag{2.7}$$

Using (2.7), we can then discretize SDE (2.4-2.5) in the following way:

$$(S_i)_m^0 = s_i^0, \tag{2.8}$$

$$(S_i)_m^{n+1} = (1 + (r - \delta_i)\Delta t) \, (S_i)_m^n + \sigma_i \, (S_i)_m^n \, (\Delta W_i)_m^n \, . \tag{2.9}$$

Equations (2.7-2.9) allow us to construct M simulated trajectories of the underlying assets in time between the time points $t_0 = 0$ and $t_N = T$ while only requiring the generation of standard normal random realizations, which can be done with any useful pseudo random generator. For the rest of this paper we use the built in generator of the *Math* library in python to generate these realizations.

Now that we have discretized the SDE and established a method to construct realizations of the underlying price trajectories, we can also discretize the BSDE through an Euler discretization method in the following way:

$$c\left(\vec{S}_m^{n+1}, t^{n+1}\right) = (1 + r\Delta t)c\left(\vec{S}_m^n, t^n\right) + \sum_{i=1}^{d} \sigma_i \left(S_i\right)_m^n \frac{\partial c}{\partial s_i}\left(\vec{S}_m^n, t^n\right)\left(\Delta W_i\right)_m^n . \tag{2.10}$$

As established in **Theorem 2.1** the assumption is made that the asset is not exercised in the time space between the previous time point and the current one. However, if we allow the option to possibly be exercised at some point after time t^{n+1}, then we can replace the future continuation value on the left hand side of (2.10) by the actual option value. If we then bring the future option value to the other side of the equation, we can define the residual of the discretized BSDE equation

$$R\left[c^n\right]_m = (1 + r\Delta t)c^n\left(\vec{S}_m^n\right) + \sum_{i=1}^{d} \sigma_i \left(S_i\right)_m^n \frac{\partial c^n}{\partial s_i}\left(\vec{S}_m^n\right)\left(\Delta W_i\right)_m^n - v^{n+1}\left(\vec{S}_m^{n+1}\right). \tag{2.11}$$

Here we utilize the functional shortcut where $v^n(\vec{s}) \equiv v(\vec{s}, t^n)$. The solution of this discretized equation at any single time step requires the finding of a d-dimensional function $c^n(\vec{s})$ where it and all of its first derivatives satisfy (2.11) in such a way that $R\left[c^n\right]_m = 0$ for all $m = 0, \cdots, M$. This problem is extremely difficult to solve analytically, so we must rely on some numerical methods to allow us to construct an approximation to c^n which minimizes the residual as much as possible. Chen and Wan [4] define this approximation to the continuation function as y^n and attempt to solve the residual problem in a least squares sense, that is, they wish to find the function y^n which satisfies

$$c^n \approx (y^n)^* \equiv \arg\min_{y^n} \left(\sum_{m=1}^{M} R\left[y^n\right]_m^2 \right). \tag{2.12}$$

All of these ideas are compiled together and Algorithm 1 defines the most general form of how to compute the solution to the given problem. This algorithm will be refined multiple times further on in this paper. This algorithm is fairly easy to understand, and the problem now becomes how do we solve the least squares minimization problem (2.12). In [4], Chen and Wan utilize a system of Neural Networks which learn to solve this minimization problem and output the difference between the value of the option at the same price vectors, but two subsequent points in time. This method is discussed in full in the next chapter.

Algorithm 1 General Algorithm to Solve the Pricing Problem

PARAMETERS

M: the number of individual underlying price paths/trajectories

N: the number of time steps

initialize the current underlying asset prices $\{\vec{S}_m^0 \equiv \vec{s}^0 | m = 1, \cdots, M\}$

for $n = 1, \cdots, N$ **do**

Generate the underlying M price paths using equations (2.7-2.9) $\{\vec{S}_m^{n+1} | \vec{S}_m^n\} \forall m$

end for

Initialize the terminal values of the options $v^N(\vec{S}_m^N) = f(\vec{S}_m^N) \forall m$

for $n = N-1, \cdots, 0$ **do**

Solve the least squares minimization problem (2.12) for $(y^n)^*$

Calculate the value of the option $v^n(\vec{S}_m^n) = \max\left[y^n(\vec{S}_m^n), f(\vec{S}_m^n)\right] \forall m$

end for

Chapter 3

Previous Solutions to the Problem

As discussed in the previous chapter, finding a solution to the least-squares minimization problem (2.12) can be complicated. However, some known methods have been discussed in previous literature from Longstaff and Schwartz [18], Kohler [16] as well as Chen and Wan [4]. The Longstaff-Schwartz method (also analyzed by Kohler) is derived from transitioning the functional space of (2.12) to a parameter space, which is much easier to work with. It defines the wanted function y^n as a parameterized degree-χ polynomial which is fit to the regression function so as to satisfy the minimization problem. However the solution of this method requires the consideration of an extensive monomial basis of size $\binom{d+\chi}{d} \approx \frac{1}{\chi!}d^\chi$. While practical applications select $\chi \ll d$, it has been shown theoretically that convergence of the method requires $\chi \to \infty$ ([18] and Stentoft [21]), resulting in an incomputable problem for higher dimensions. The intricacies of the Longstaff-Schwartz are outside the scope of this report, however we discuss the computational cost of this method in Chapter 5 and direct interested readers to [18].

The paper by Chen and Wan [4] solves the minimization problem (2.12) by constructing a neural network which learns the dynamics of the residual equation and concurrently outputs both the option values and deltas. They also utilize this solution in the form of two related algorithms, the first of which from this point on we will refer to as their 'original' algorithm as defined in 4.1 of [4]. The second form of the algorithm will be referred to from this point on as their 'efficient' algorithm. Chen and Wan's efficient algorithm is required for the construction of the parallel algorithm so for the scope of this paper we will discuss their efficient algorithm and generally bypass the original construction of the algorithm, however, interested readers are directed to read the publication of Chen and Wan to see the original algorithm.

3.1 Chen and Wan's Neural Network Algorithm

Recall that for the discretization defined in Chapter 2, we utilize M Monte Carlo price simulations with N time steps which evenly segment the temporal domain between the temporal limits $t_0 = 0$ and $t_N = T$. However, instead of stepping backwards one temporal step at a time, which is how the original algorithm works from Chen and Wan [4], their efficient algorithm works by using two levels of discretization. We denote an additional parameter J which we will call a 'milestone step size', and thus we also denote the first level of temporal discretization as the set of 'milestone time points' which can mathematically be defined by the set $\Psi = \{t_n | n = Ji, i = 0, \cdots, \frac{N}{J} - 1\}$. Note that the milestone points begin at t_0 and occur at every j^{th} time point in the discretization, but does not include the final time point $t_n = T$. This choice to not include the final time point is theoretically ornamental, however, it does make some of the calculations easier in later analysis. The choice of milestone points is further supported by the fact that at the terminal time, we can explicitly calculate the value of the option strictly by the payoff function of the price vectors at that time, so this time point doesn't require any analysis in most situations. The second level of discretization is what we will define as 'fine time points' which include all time points in the discretization which are not designated as milestone points, i.e., $\Psi' = \{t_n | t_n \notin \Psi\}$.

The efficient algorithm by Chen and Wan utilizes a sequence of N neural networks which are each responsible for learning the change in option value between the same price vector at two different time points. The neural network works by learning a basis for a nonlinear paramaterization which minimizes a given loss function over a set of training points, and thus learns the optimal basis of parameters at the end of the training time (Goodfellow *et al.* [12]). Denote the set of N neural networks as $\{y^n(\vec{s}, \Omega^n) | n = N-1, \cdots, 0\}$ which outputs the value of the option at price vector $\vec{s}$ based on the value of the same price vector at some future time. Chen and Wan use this definition through the use of the following equations:

$$y^N(\vec{s}) = f(\vec{s}), \tag{3.1}$$

$$y^n(\vec{s}; \Omega^n) = y^{n+j}(\vec{s}; \Omega^{n+j}) + j\Delta t F(\vec{s}; \Omega^n). \tag{3.2}$$

Here j is defined in such a way that if $t_n \in \Psi$ then $j = J$ and if $t_n \notin \Psi$ then $n + j$ is the minimum value of $k > n + j$ such that $t_k \in \Psi$. Put more simply, this means that we traverse backwards across the milestone points from the terminal time, and at each milestone point, we have $J - 1$ networks which branch off to the fine time points in between the two subsequent milestone times.

3.1.1 Neural Network Architecture

In equation (3.2), $F(\vec{s}, \Omega^n)$ represents the output of the neural network between times $n+j$ and n, which has learned the difference between the two approximations of the continuation functions at different times, and Ω^n represents the set of learnable parameters of the neural network. Equation (3.2) is also a useful definition for the implementation of the neural network because even before any training occurs, we notice that the error of the option value at time n is only of the order $O(\Delta t)$ with respect to the previous function value. The network F is implemented as an L-layer feedforward network with layers indexed by $l = 0, \cdots, L$, let superscripts with square brackets denote the index of the layer of the network and thus the input layer of the network is $\vec{x}^{[0]} = \vec{s} \in \mathbb{R}^d$. Chen and Wan define the rest of the network in [4] as follows:

- For layers, $l = 1, \cdots, L$,

$$\vec{z}^{[l]} = \mathbf{W}^{[l]} \times \vec{x}^{[l-1]}, \tag{3.3}$$

$$\vec{h}^{[l]} = \text{bnorm}(\vec{z}^{[l]}; \vec{\beta}^{[l]}, \vec{\gamma}^{[l]}, \vec{\mu}^{[l]}, \vec{\sigma}^{[l]}), \tag{3.4}$$

$$\vec{x}^{[l]} = \max(\vec{h}^{[l]}, 0), \tag{3.5}$$

  where

$$\text{bnorm}(\vec{x}; \vec{\beta}, \vec{\gamma}, \vec{\mu}, \vec{\sigma}) \equiv \vec{\gamma}\frac{\vec{x} - \vec{\mu}}{\vec{\sigma}} + \vec{\beta} \tag{3.6}$$

  is the batch normalization operator with moving averages and standard deviations of batches $\vec{\mu}^{[l]}, \vec{\mu}^{[l]} \in \mathbb{R}^{d^{[l]}}$ and $\vec{\gamma}^{[l]}, \vec{\beta}^{[l]} \in \mathbb{R}^{d^{[l]}}$ are trainable scales and offsets. Finally $\mathbf{W}^{[l]} \in \mathbb{R}^{d^{[l]} x d^{[l-1]}}$ is a trainable matrix of values.

- For the output layer

$$F(\vec{s}; \Omega^n) = \vec{\omega} \times \vec{x}^{[L]} + b, \tag{3.7}$$

  where $\vec{\omega} \in \mathbb{R}^{d^{[L]}}, b \in \mathbb{R}$ are trainable weights and bias values.

Finally, Chen and Wan suggest a final addition to the output of the network architecture in the form of a trainable parameter α which helps to expand the functional basis of the network so it can better represent a wider variety of functions. This wider functional basis of the neural network helps to better fit the training data as suggested by Goodfellow [12]:

$$y^n(\vec{s}; \Omega^n) = \alpha^n \left[y^{n+j}(\vec{s}; \Omega^{n+j}) + j\Delta t F(\vec{s}; \Omega^n) \right]. \tag{3.8}$$

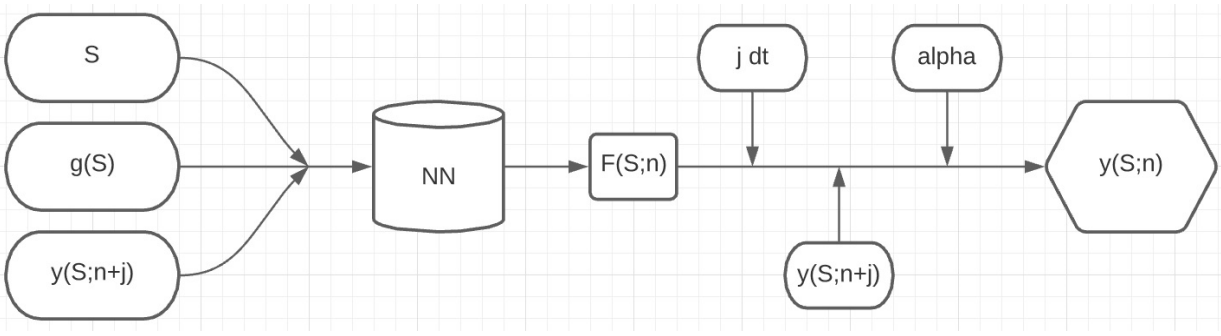

Figure 3.1: Overview of the neural network architecture separated into 3 sections: The input layer, the feedforward network, and the linear relation between continuation values

In [4], two additional inputs to be submitted in the input layer of the network framework. The first is the internal function of the option payoff $g(\vec{s})$, the heuristic idea behind this is it allows the network to intrinsically learn the current payoff value of the function and learn how that affects the option continuation value. The reason $g(\vec{s})$ is inputted and not $f(\vec{s})$ is due to the maximum operator included in f. This means that f can be uniquely determined given g, but not the other way around. Therefore, inputting g gives the network more information to learn from. Secondly, they suggest inputting the previous continuation value of the option for a very straightforward reason. Over smaller time steps, the difference between continuation values at different times should not change a large amount, so inputting the previous continuation value gives a good 'initial value' for the network to learn from.

Figures (3.1) and (3.2) provide visual examples of the architecture of the networks and the flow of outputs across time steps respectively (3.2 also appears in Chen and Wan's paper [4] in section 4.4). Note that all the previous equations only relate to the actual continuation values of the option and make no mention of the deltas of the values which are a requirement to solve the minimization problem (2.12). However, the deltas can be extracted from the previous equations by taking the gradient of equation (3.2). It is assumed that the deltas from future time steps have already been calculated so all that is required is to calculate the deltas of the output value $\vec{\nabla} F$, or the change in the output of the network with respect to the input price vector values. These values can be calculated simply in the tensorflow implementation of the code through the *tf.gradients()* functions and is incorporated into the architecture of the network.

One final feature of the architecture to bring attention to is that equation (3.2) relates the continuation value of the option at different times, but with respect to the same un-

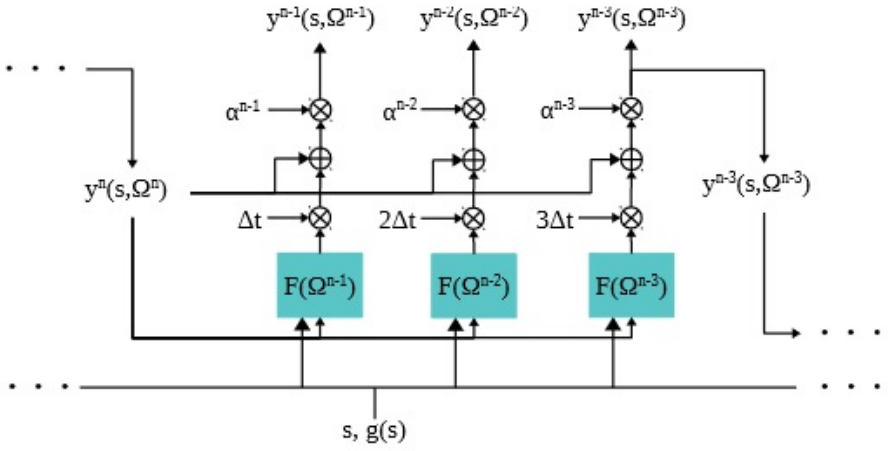

Figure 3.2: The efficient neural network framework suggested by Chen and Wan, where J = 3.

derlying asset price vector. However, there is no guarantee that we will have the same price vector appear more than once in any part of the Monte Carlo paths, especially as the dimensionality of the option increases. To properly complete the algorithm, Chen and Wan implement a solution where we initialize the value of the option at every time point along every simulated path by the payoff function at the terminal time $Y_m^n = f(\vec{S}_m^n)$, then as we step backwards in time, you update the values of the options according to two rules. If $t_n \notin \Psi$, then $Y_m^\nu = y^n(\vec{S}_m^\nu; (\Omega^n)^*), \forall m$, and if $t_n \in \Psi$, then $Y_m^\nu = y^n(\vec{S}_m^\nu; (\Omega^n)^*), 0 \leq \nu \leq n, \forall m$. Put more simply, if we are arriving at a milestone time point, then we need to carry forward the continuation values for all time points that happen before t_n. In contrast, if we are branching off to a fine time point, then we only need to update the option values there for the final time and no other time points are dependent on the values there. Once a continuation value has been updated for the final time, we can calculate the value of the option by simply taking the maximum of the continuation value $y^n(\vec{S}_m^n; (\Omega^n)^*)$ and the payoff value $f(\vec{S}_m^n)$. Here $(\Omega^n)^*$ denotes the optimal set of trained parameters reached after training.

3.1.2 Network Training

After finishing a previous time step, we must train the neural network to solve the minimization problem through training. To do so we require the set of training inputs which are as follows:

$$\{\vec{S}_m^n, \Delta\vec{W}_m^n, v^{n+1}(\vec{S}_m^{n+1}), g(\vec{S}_m^n), y^{n+j}(\vec{S}_m^n; (\Omega^{n+j})^*), \vec{\nabla} y^{n+j}(\vec{S}_m^n; (\Omega^{n+j})^*) | \forall m\}. \tag{3.9}$$

Of course the first, second and forth values are known or can be calculated immediately as a result of the Monte Carlo simulated price paths, and the other three values have been learned at previous time steps. The first three values are what is required to be inputted into the minimization problem equation (2.12) while the final three values are requirements of the neural network input layer or architecture to calculate training outputs at the current time step. By calculating he last three values and the price vector through the neural network, we get training outputs of the form $\{y^n(\vec{S}_m^n; \Omega^n), \vec{\nabla} y^n(\vec{S}_m^n; \Omega^n)\}$. Now we finally have everything required to properly train the neural network. We utilize the minimization equation (2.12) as the loss function of the network, i.e., given the set of training inputs and outputs, we can measure the current accuracy of the network by plugging these values into the following:

$$L\left[\Omega\right] \equiv \sum_{m=1}^{M} \left[(1 + r\Delta t) y^n \left(\vec{S}_m^n\right) + \sum_{i=1}^{d} \sigma_i \left(S_i\right)_m^n \frac{\partial y^n}{\partial s_i} \left(\vec{S}_m^n\right) \left(\Delta W_i\right)_m^n - v^{n+1}\left(\vec{S}_m^{n+1}\right) \right]^2. \tag{3.10}$$

Once we have a network architecture and a proper residual loss function, we can now adjust the trainable parameters of the model through a form of gradient descent to try to minimize the loss generated by the network. As a result of this method, the network learns to force this residual as close to zero as possible and as a byproduct solve the minimization problem at the same time. There are many useful ways to achieve this but the original algorithm utilizes the Adam optimizer (Kingma and Ba [15]). From this, we gain the set of optimized parameters $(\Omega^n)^* \equiv \arg\min_{\Omega^n} L[\Omega^n]$.

In [4], the authors discuss a number of other intricate topics with relation to improving upon the original algorithm. These topics include weight reuse between networks, correcting the training inputs, utilizing network ensembles and the calculation of the asset value at time $t_0 = 0$. These topics are omitted here, but interested readers are directed to read the original paper as we continue to utilize these methods in the parallel form which is introduced in the next chapter. Algorithm 2, which also appears in [4], provides a step by step walkthrough of completing Chen and Wan's efficient algorithm in the manner that was shown in this chapter.

Algorithm 2 Chen and Wan's Solution to the Pricing Problem

PARAMETERS

M: the number of individual underlying price paths/trajectories
N: the number of time steps
J: the size of milestone time steps

initialize the current underlying asset prices $\{\vec{S}_m^0 \equiv \vec{s}^{\,0} | m = 1, \cdots, M\}$

for $n = 1, \cdots, N$ **do**
  Generate the underlying M price paths using equations (2.7-2.9) $\{\vec{S}_m^{n+1} | \vec{S}_m^n\} \forall m$
end for

Initialize the terminal values of the options and their deltas
$Y_m^n = f(\vec{S}_m^n)$ and $Z_m^n = \vec{\nabla} f(\vec{S}_m^n) \forall m, n$
Initialize $v^N(\vec{S}_m^N) = Y_m^N \forall m$

for $n = N-1, \cdots, 0$ **do**
  Initialize the neural network $y^n(\vec{s}; \Omega^n)$ defined by (3.1-3.8)

  Training: minimize the loss function (3.10) yielding the optimal trained network $y^n(\vec{s}; (\Omega^n)^*)$

  if $(N-n) \mod J = 0$ **then**
    (All upcoming time steps) Overwrite the options and deltas:
    $\{Y_m^\nu = y^n(\vec{S}_m^\nu; (\Omega^n)^*), 0 \le \nu \le n, \forall m\}$
    $\{Z_m^\nu = \vec{\nabla} y^n(\vec{S}_m^\nu; (\Omega^n)^*), 0 \le \nu \le n, \forall m\}$
  else
    (All upcoming time steps) Overwrite the options and deltas:
    $\{Y_m^n = y^n(\vec{S}_m^n; (\Omega^n)^*), \forall m\}$
    $\{Z_m^\nu = \vec{\nabla} y^n(\vec{S}_m^\nu; (\Omega^n)^*), \forall m\}$
  end if

  update $\{v^n(\vec{S}_m^n) = \max(Y_m^n, f(\vec{S}_m^n)) | \forall m\}$

end for

RESULT: Sample of option prices Y_m^n and deltas Z_m^n on the entire domain

Chapter 4

The Parallel Algorithm

Parallel processing, and more specifically its implementation in the field of financial mathematics is not new. We see implementations of parallel methods in finance as early as 2002 in the work of Bal and Maday [2] and further advancements were made in the field over the next two decades including by Falgout *et al.* [8]. The issue with attempting to apply these previous works to the problem found here is that they are developed and applied to problems involving PDE formulations. While nonlinear PDE solutions do exist for American option pricing problems, for large-dimension problems, these PDE forms become extremely complex and the iterative methods required to solve them become computationally infeasible. The second issue with PDE solutions is that they don't also output the delta values for the option, and thus may be less desirable than the method proposed here. Parallel processing methods tend to work on many different levels of discretization, the theory is that you can work on a sparse grid to get an initial approximation of the solution to the problem, then work an a finer grid while correcting the solution based on the initial approximation. This type of method does not appear as if it can be feasibly applied to this problem for a few reasons. Firstly, on every finer grid, you must generate a new set of price vectors at different times making it difficult to apply any correction factors. Secondly, the algorithm must begin at the terminal time due to the nature of the continuation function, so it's difficult to shortcut running the algorithm on a smaller grid once the initial approximations have been found. We note that a backward recursive parallel method for pricing options was developed by Wan *et al.* [23] in 2006 which is based off of existing low discrepancy mesh methods and applies ideas from quasi-Monte Carlo and stochastic mesh techniques. The exact difficulty of applying these types of methods to this problem can be a future work on it's own, but we propose a new parallel processing method by segmenting the original problem into a form which allows processors to work independently from one

another to save time and memory.

4.1 Partitioning the Previous Algorithm

The construction of the proposed parallel algorithm begins with analyzing the dependence of the value of the option at a general price vector and time, with respect to all the other time points used in the discretization. Recall equation (3.2) $y^n(\vec{s};\Omega^n) = y^{n+j}(\vec{s};\Omega^{n+j}) + j\Delta t F(\vec{s};\Omega^n)$ where if $t_n \in \Psi$ then $j = J$, and if $t_n \notin \Psi$ then $n+j$ is the minimum value of $k > n+j$ such that $t_k \in \Psi$. We can extend this formulation to write the value of the option at n as a function of all previous option milestone networks

$$y^n(\vec{s};(\Omega^n)^*) = y^N(\vec{s}) + j\Delta t F(\vec{s};(\Omega^n)^*) + \sum_{k|t_k\in\Psi,k\geq n+j} J\Delta t F(\vec{s};(\Omega^k)^*). \tag{4.1}$$

This is constructed by recursively applying (3.2) to the y^k on the right hand side of (3.2) until you reach the terminal value of $y^N(\vec{s})$. It is clear now by observing the summation on the right hand side of (4.1) that the continuation value of option y^n can be reached by only traversing through the milestone networks with time points greater than n, and finally by traversing the fine network n. In this sense, the solutions to the algorithm for $\{t_k|t_k > t_n, t_k \notin \Psi\}$ are irrelevant in obtaining the solution at time point n. Of course this is only true under the construction of the previous algorithm defined in Chapter 3, but this fact is quintessential in segmenting the work across multiple processors and thus resulting in time saves.

For further explanation, consider the following thought experiment: Imagine we wish to solve for the values and deltas of the option at two times $t_k, t_{k+1} \notin \Psi$, we will consider the example of a single processor and two individual processors completing the algorithm for this task, with the parallel processors being individually responsible for one of the two fine points. Firstly, we allow all 3 processors to independently train and compute the networks across each of the milestone time steps until arriving at the final milestone point before the pair of fine points we're interested in calculating. Now, each of the pair of processors can train and calculate each of the finer networks concurrently whereas the single processor would need to complete these two tasks sequentially and would take twice as long for this step, resulting in a time save for the parallel processors. This is not the only time the parallel algorithm would save however, at each of the milestone steps, the single processor would need to update the option continuation values and deltas at two future fine time

points, whereas each parallel processor would only need to complete this task for the single fine point it's responsible for.

The thought experiment gives a general idea of the process involved with constructing the proposed parallel algorithm. If we divide up the set of time points $\{t_k | 0 \leq k < N\}$ across the number of processors available to us, K, we can allow each of the independent processors to solve the option values and deltas only in the milestone and fine time points that we have designated it to be responsible for. Designate the K processors as $\{P_q | q = 0, \cdots, K-1\}$, and if we naively divide up the N time steps sequentially amongst the K processors, then processor P_q is responsible for calculating the time points

$$\{t_{\frac{N}{K}q+l} | l = 0, \cdots, \frac{N}{K} - 1\}. \tag{4.2}$$

We then allow each of the processors to independently solve the algorithm in Chapter 3, but only for the time points it has been given responsibility for, as well as all milestone times prior to its earliest time. In this case, each processor P_q individually needs to solve the algorithm along the set of points

$$\{t_{q_h}\} = \{t_{\frac{N}{K}q+l} | l = 0, \cdots, \frac{N}{K} - 1\} \cup \{t_j | t_j \in \Psi, j \geq \frac{N}{K}q\}, \tag{4.3}$$

where $h = 0, \cdots, H$, and H may vary depending on the size of N, K, J.

Once all of the processors have completed their algorithm, we have each of them send the option prices and deltas on the solved set of time points it was responsible for to a single processor and paste all of the time points together into a complete space-time solution.

4.2 Improving the Parallel Algorithm

The original naive partition of the set of time points does result in temporal wall clock and memory cost savings over the efficient algorithm by Chen and Wan in [4], however it is not the best approach to take and we can improve the amount of savings to be made. We begin improving our assumptions with the Monte Carlo simulations of the price path and realize that for each individual processor, the solution of the algorithm on the finer points which the processor is not responsible for do not influence the solution that this processor will compute. This also means that the trajectories of the Monte Carlo price paths on those irrelevant points are not important and can be disregarded, i.e. we don't need to simulate these price vectors to begin with and each individual processor only needs to simulate price

trajectories along the time points it is responsible for. Therefore, we can adjust equation (2.7-2.9) to only simulate prices along these time points for each individual processor, i.e., given processor P_q, simulate trajectories

$$(\Delta W_i)_m^n = \sum_{j=1}^{d} L_{ij} \, (\Phi_j)_m^n \, \sqrt{t_{q_{n+1}} - t_{q_n}}, \tag{4.4}$$

$$(S_i)_m^0 = s_i^0, \tag{4.5}$$

$$(S_i)_m^{n+1} = \left(1 + (r - \delta_i)(t_{q_{n+1}} - t_{q_n})\right) (S_i)_m^n + \sigma_i \, (S_i)_m^n \, (\Delta W_i)_m^n \, , \tag{4.6}$$

where the set t_{q_h} is as defined in equation (4.3). This clearly saves both memory and wall clock time (explored in Chapter 5) since we need to simulate and store fewer price trajectories than in the previous algorithm.

We can make a second large improvement over the algorithm by coming up with a smarter way of partitioning the time points among the K processors. Let's begin by assigning time t_0 to processor P_0, and look at what happens in this case. Under this assumption, this processor would need to traverse all $\frac{N}{J}$ milestone steps to arrive at time $t_0 = 0$ and calculate the initial value of the option. Note however that in order to complete the algorithm, this processor has already computed the solution to the problem at every milestone time along the way, making it redundant to assign these times to any other processors. This means that we only need to partition the $N - \frac{N}{J}$ fine time points among each of the K processors, rather than the entirety of the set $\{t_k\}$. It's initially unclear as to whether or not this leads to a reduction in wall clock time or memory requirements, however we can say that this immediately reduces the upper bound on the number of processors required to see substantial gains in the algorithm since we at most only require the use of $N - \frac{N}{J}$ processors instead of N, which reduces with the size of our milestone step J.

We make one more assumption with respect to the parallel algorithm that doesn't result in computational saves, but does result in a more even load split among the K processors. We will say that regardless of the fine time points which are given as a responsibility to each of the processors, we will force all of the processors to compute all of the milestone times and finish at t_0. The reasoning behind this has less to do with the fact that it results in a temporal or memory save (in fact it can be shown that there is an increase in both) but in the field of parallel processing, it is generally considered inefficient to have some processors complete their algorithms earlier and have to sit around waiting for others to finish. The amount of wall clock time increase is small since one processor needs to compute all the

milestone times anyways and will generally dominate the overall time as shown in Chapter 5. The increase in memory can be justified by the fact that forcing a maximum step size, J, in the price path trajectories helps to limit the variation of the underlying paths and keep the number of paths required for accurate results smaller. Forcing this new rule on the processors helps to force all processors to finish their algorithms closer to one another and reduce the amount of down time of each of them. We note however that this load split could also be done through a more clever partition of the fine time points since the fine times closer to t_0 hold a larger weight load than later times (also shown in Chapter 5), and could be the focus of some future work. We summarize the proposed parallel algorithm as well as the noted improvements in Algorithm 3.

Algorithm 3 Proposed Parallel Solution to the Pricing Problem

PARAMETERS

M: the number of individual underlying price paths/trajectories
N: the number of time steps
J: the size of milestone time steps
K: the number of processors used

for $q = 0, \cdots, K-1$ concurrently **do**
initialize the current underlying asset prices $\{\vec{S}_m^0 \equiv \vec{s}^{\,0} | m = 1, \cdots, M\}$

for n s.t. $t_n \in \{t_{q_h}\}$ **do**
Generate the underlying M price paths using equations (4.4-4.6) $\{\vec{S}_m^{n+1} | \vec{S}_m^n\} \forall m$
end for

Initialize the terminal values of the options and their deltas
$Y_m^n = f(\vec{S}_m^n)$ and $Z_m^n = \vec{\nabla} f(\vec{S}_m^n) \forall m$, $n : t_n \in \{t_{q_h}\}$
Initialize $v^N(\vec{S}_m^N) = Y_m^N \forall m$

for n s.t. $t_n \in \{t_{q_h}\}$ **do**
Initialize the neural network $y^n(\vec{s}; \Omega^n)$ defined by (3.1-3.8)

Training: minimize the loss function (3.10) yielding the optimal trained network $y^n(\vec{s}; (\Omega^n)^*)$

if $(N-n) \mod J = 0$ **then**
(All upcoming time steps) Overwrite the options and deltas:
$\{Y_m^\nu = y^n(\vec{S}_m^\nu; (\Omega^n)^*), 0 \le \nu \le n, \forall m, t_\nu \in \{t_{q_h}\}\}$
$\{Z_m^\nu = \vec{\nabla} y^n(\vec{S}_m^\nu; (\Omega^n)^*), 0 \le \nu \le n, \forall m, t_\nu \in \{t_{q_h}\}\}$
else
(Current time step) Overwrite the options and deltas:
$\{Y_m^n = y^n(\vec{S}_m^n; (\Omega^n)^*), \forall m\}$
$\{Z_m^\nu = \vec{\nabla} y^n(\vec{S}_m^\nu; (\Omega^n)^*), \forall m\}$
end if

update $\{v^n(\vec{S}_m^n) = \max(Y_m^n, f(\vec{S}_m^n)) | \forall m\}$

end for
Send the completed sample of option prices and deltas on fine time points to processor 0
end for

RESULT: Processor 0 compiles sample of option prices Y_m^n and deltas Z_m^n on the entire domain

Chapter 5

Computational Cost and Efficiency

This chapter analyzes the computational cost required in both memory requirements and the wall-clock time required to complete each of the algorithms. We also study the efficiency of the proposed parallel algorithm, i.e., how much wall-clock time is saved as extra processors are used in the proposed parallel algorithm.

We will begin each section of this chapter by first establishing the baseline amount of memory and time cost of the Longstaff-Schwartz method [18] and will compare it to the original and efficient algorithms proposed by Chen and Wan [4] (the algorithms are described in sections 4.1 and 4.4 respectively of that publication). Much of this analysis was also presented in the paper by Chen and Wan, but we restate it as a way of establishing a baseline to compare the costs of their algorithms with the parallel algorithm proposed in this paper.

Recall from earlier and the works of Longstaff and Schwartz [18] and Kohler [16] that the Longstaff-Schwartz method utilizes a degree-χ polynomial, which requires the consideration of a monomial basis of the form:

$$\phi_\chi(\vec{s}) \equiv \{s_1^{a_1} s_2^{a_2} \cdots s_d^{a_d} | a_1 + a_2 + a_d \leq \chi\}. \tag{5.1}$$

In practical applications, one chooses $\chi \ll d$ to keep the cost of the approach within a computable range, and the size of the monomial basis is then $\binom{d+\chi}{d} \approx \frac{1}{\chi!} d^\chi$.

5.1 Memory Cost

An important consideration when discussing the effectiveness of the proposed algorithm is the amount of memory that is required to complete the algorithm. It is known that the pricing problem of American options is subject to the curse of dimensionality, i.e., the amount of memory required is exponential in the dimension of the option. As shown in the paper by Chen and Wan [4], for larger-scale problems ($d \geq 100$) the Longstaff-Schwartz method requires an excessive number of basis vectors and computers run out of memory. We show that the proposed parallel algorithm further saves on memory requirements for each of the individual processors as compared to the original and efficient algorithms proposed by Chen and Wan.

5.1.1 Longstaff-Schwartz Method

For the Longstaff-Schwartz Method, we first are required to store the d-dimensional price vectors along each of the M price paths at all N time steps, resulting in an initial memory cost of NMd floating point values $\{\vec{S}_m^n | \forall n, m\}$. Secondly, while completing the actual algorithm, at each time step we also need to store the current values of the $\frac{1}{\chi!}d^\chi$ monomial basis functions for all paths, and the value of the option at each of the price paths in the previous time point. In total at each time point, we must store $\{\phi_\chi(\vec{S}_m^n), y^n(\vec{S}_m^n) | \forall m\}$. Therefore, the Longstaff-Schwartz method requires the storage of $NMd + \frac{1}{\chi!}d^\chi + M = M(Nd+1) + \frac{1}{\chi!}d^\chi$ floating point values. Recall that, although for practical applications we choose a value of $\chi \ll d$, for convergence, the method requires $\chi \to \infty$, resulting in a rapidly growing memory cost for larger-dimensional option problems.

5.1.2 Chen and Wan's Methods

Chen and Wan's original and efficient algorithms both utilize the same amount of total computer memory. These algorithms replaces the monomial basis of the Longstaff-Schwartz polynomial with the training values that are used to train the neural networks, and the trainable parameters of the neural networks, between different time steps. For the trainable parameters we assume the nueral network contains L hidden layers with constant width z. Given $d+2$ input nodes and the final trainable parameter α, a total of $(d+3)z + (L-1)(z^2+z) + z + 2 = (L-1)z^2 + (d+L+2)z + 2$ floating point trained parameters must be stored. Both methods also need to store the NMd price point floating values along all price paths and at each time step. Since the neural networks need to be trained, the

algorithm finally requires the storage of the training input values from the previous time step $\{y^{n+\eta}(\vec{S}_m^n), \vec{\nabla} y^{n+\eta}(\vec{S})_m^n | \forall m\}$. The algorithm also needs to store the training outputs $\{y^n(\vec{S}_m^n), \vec{\nabla} y^n(\vec{S})_m^n | \forall m\}$ to train the neural networks via the loss function. The training inputs and outputs results in another $2(M + Md)$ floating point values, resulting in a total memory cost of $(L-1)z^2 + (d+L+2)z + 2 + NMd + 2(M + Md)$ floating point values.

We can see that Chen and Wan provided a method with linear memory cost in d, providing significant gains in this regard over Longstaff and Schwartz, which required a memory availability that was not only polynomial in d, but also required a large polynomial dimension χ to ensure convergence to the correct solution. This gain is significant because it allows traders to calculate values and deltas for options with significantly higher dimensionality than shown in previous literature. We see this in the numerical results published by Chen and Wan which saw out-of-memory errors presenting themselves for options of dimensionality ≥ 100 using the Longstaff-Schwartz method, but that their method is able to compute.

5.1.3 Proposed Parallel Algorithm

The algorithms proposed by Chen and Wan improved upon Longstaff and Schwartz by reducing the memory cost required in terms of the number of floating point values to be stored during the solving of the continuation function. The proposed parallel algorithm improves upon Chen and Wan in terms of the number of total price path points that need to be saved by each of the individual processors. Recall that we partition the grid of fine points and evenly distribute them across the processors that are available to the algorithm. This means that each processor no longer needs to save the underlying asset prices on the entire space-time and only needs to save the prices at the $\frac{N}{J}$ milestone points, and at the $\frac{N-\frac{N}{J}}{K}$ fine time points that that processor is responsible for calculating. Therefore, instead of N time points to be saved, each processor only needs to save $\frac{N+\frac{N}{J}(K-1)}{K}$ time points of asset prices, resulting in a memory cost of $\frac{N+\frac{N}{J}(K-1)}{K}Md + 2(M + Md)$. Finally, this parallel method requires saving the same number of trainable parameters as the algorithm by Chen and Wan, leaving a total memory cost per processor of $(L-1)z^2 + (d+L+2)z + 2 + \frac{N+\frac{N}{J}(K-1)}{K}Md + 2(M + Md)$.

We note that the total cost across the entirety of the processors together is higher than Chen and Wan's algorithm since all of the milestone points are saved across each of the individual processors. However, distributing the work of the fine time points across the processors individually requires a smaller amount of allocated memory availability than

what is required for one processor running the entirety of either one of Chen and Wan's algorithms. As discussed previously, smaller memory requirements allow for higher dimensionality, but also allow for more time steps to be taken since the work is distributed across many more processors. This allowance of more time steps can help offset the accuracy losses sustained in the delta hedging process from larger milestone steps, potentially making this algorithm more enticing to investors and market traders.

5.2 Computational Time Cost

Alongside the memory cost that was discussed throughout the previous section, another important metric to consider is the computational wall clock time that is required to complete the algorithm. This is important to traders and market researchers in the real world who wish to trade and hedge assets. Algorithms that can be completed in a quicker time frame result in options being able to be traded more often, and hedging strategies being updated on a more regular basis. This results in a less risky portfolio and more market information in a quicker time frame, which is always good in the eyes of market investors.

The mathematical formulae of the wall clock time requirements of each of the methods require a bit more in-depth analysis than the memory analysis seen previously in this paper. While the analysis of Longstaff-Schwartz and their own methods was shown by Chen and Wan [4], this paper will show the same analysis in a slightly different way in an attempt to better show how exactly the proposed parallel method results in faster execution times.

5.2.1 Longstaff-Schwartz Method

As shown both by Longstaff and Schwartz [18] and Chen and Wan [4], under the assumption that the standard normal equation is used for solving the regression problem posed earlier in this paper, the Longstaff-Schwartz method results in a computational time that is $O\left(NM\left(\frac{1}{\chi!}d^{\chi}\right)^2\right) = O\left(NMd^{2\chi}\right)$. Of note, this method is worse than quadratic in the dimensionality of the option, and we will see that Chen and Wan's algorithm solves this issue asymptotically.

5.2.2 Chen and Wan's Methods

To analyze the amount of computational time saved by Chen and Wan in more detail than [4], we will first lay out three assumptions that will allow us to describe the amount of time required to complete the algorithm. We will begin by assuming that the time for a processor to simulate one time step of the Monte Carlo paths is constant, γ, secondly, the time to train one of the neural networks between two time points is a constant value, β, and the time required to update the option values at a single time point, i.e., to run all M asset price vectors of a single slice of time through the neural network is also a constant, α. These assumptions are coarse but since the number of training inputs and outputs is constant for every neural network, all networks are of the same architecture, and all computations are done on the same computer, it is a reasonable assumption to make in the context of the problem. For simplicity of the analysis, we further assume that $J|N$, or that the milestone time points evenly discretize the overall finer mesh.

Under these assumptions the computational time required for the original sequential algorithm is

$$N(\gamma + \beta) + \sum_{i=1}^{N} i\alpha = N(\gamma + \beta) + \frac{\alpha}{2}\left(N^2 + N\right). \tag{5.2}$$

In this expression the first term $N\beta$ comes from the fact that there is one neural network between each of the N time points, and thus we need to train each of these networks sequentially, resulting in a total time cost of $N\beta$. The second summation term comes from the fact that after training each of the neural networks, we need to evaluate the network for all price vectors that are positioned at a time point previous to the current one. Early on in the algorithm we need to compute these updates at many time points, but as the algorithm progresses, this number shrinks by one for each step. The result shows that the computational time required is quadratic in N (Chen and Wan also show that it is quadratic in d as well).

To analyze the computational time differences between the original and efficient algorithms in Chen and Wan [4], we need to first understand the differences between the two versions. It is easier to understand the analysis by first imagining that the time points are segmented into the set of milestone times which occur every J steps and the set of fine times which are all other time points in the discretization. Since we assume that $J|N$, we can say that there are exactly $\frac{N}{J}$ milestone times and $N - \frac{N}{J}$ fine time points. For each of the milestone steps, we need to first train the neural network from the previous milestone

time to the current one, and then we must update the option values and deltas for the current time step and all previous time steps. Since the milestone times occur every J steps, then we only need to take every J^{th} term of the summation in the original algorithm. Once we have computed the milestone points, then for each fine time step we only need to train the neural network, and update the option values and deltas for that current time, and we do this for all $N - \frac{N}{J}$ fine times. Then the computational time required for the efficient algorithm in [4] is

$$\begin{aligned} &N\gamma + \frac{N}{J}\beta + \sum_{i=1}^{\frac{N}{J}} (Ji)\,\alpha + \left(N - \frac{N}{J}\right)(\beta + \alpha) \\ =&N(\gamma + \beta) + \frac{\alpha}{2}\left(\frac{N^2}{J} + N\right) + \left(N - \frac{N}{J}\right)\alpha \\ =&N(\gamma + \beta) + \frac{\alpha}{2}\left(\frac{N^2}{J} + 3N - 2\frac{N}{J}\right). \end{aligned} \tag{5.3}$$

Here we immediately note that if we select the milestone step size of $J = 1$, then we reconstruct the computational time function for the original sequential algorithm, which makes sense as then every time point would be a milestone point and we would have no finer grid of time steps. It may not be immediately obvious, but a bit of analysis also shows that as J increases, the computational time decreases, to the point where if we took one milestone step from $t = T$ to $t = 0$ ($J = N$), the computational time cost becomes linear in N, however this would be expected to result in a high loss of accuracy.

5.2.3 Proposed Parallel Algorithm

The computational time cost of the proposed parallel algorithm continues to build upon the gains that were made by the previous algorithms, but for simplicity we make one more assumption with regards to the number of processors K that we use in the algorithm. For this analysis, we assume that $\frac{N}{J} | K$, which allows us to segment the fine time points between each of the milestone times evenly across processors, and each processor is only responsible for a set of fine time points confined between two subsequent milestone points. This also means that the number of processors is required to be an integer value in the range $\frac{N}{J} \leq K \leq N$, which is not a theoretical restriction, however, we denote it in this way to ensure no processor is responsible for a set of fine time points which can be separated by any milestone points to simplify the theoretical calculations which follow. Now denote

the minimal fine time point that a single processor is responsible for as t_p^{min} where p is the index of the processor in question, we define the value $\zeta = |\{t < t_p^{max} | t = i * J\Delta t \geq t_0, i \in \mathbb{N}U\{0\}\}|$, or equivalently, ζ is the number of milestone time points which are at a time less than the fine time points that the processor is responsible for. This is important to note because the algorithm only updates the fine time values when evaluating the larger milestone times, and not the ones that come between t_0 and the fine times. Also recall that a consequence of the memory cost analysis we only need to simulate $\frac{N+\frac{N}{J}(K-1)}{K}$ price trajectories. With these additional assumptions and definitions, we can then define the computational wall clock time for any individual processor to be

$$\begin{aligned}
&\frac{N+\frac{N}{J}(K-1)}{K}\gamma + \frac{N}{J}\beta + \sum_{i=1}^{\zeta} i\alpha + \sum_{i=\zeta+1}^{\frac{N}{J}} \left(i + \left(\frac{N-\frac{N}{J}}{K}\right)\right)\alpha + \left(\frac{N-\frac{N}{J}}{K}\right)(\beta+\alpha) \\
=&\frac{N+\frac{N}{J}(K-1)}{K}\gamma + \frac{N}{J}\beta + \frac{\alpha}{2}\left(\frac{N^2}{J^2}+\frac{N}{J}\right) + \left(\frac{N-\frac{N}{J}}{K}\right)\left(\beta + \alpha\left(\frac{N}{J}-\zeta+2\right)\right). \quad (5.4)
\end{aligned}$$

While this is the clock time for any individual processor to complete their part of the algorithm, we need to define the time required for the entire algorithm as a whole. Since all processors begin at the same time, the time required for the algorithm is equal to the longest individual processor time. By analyzing the previous equations, the term inside the first summation is always greater than the term in the second summation, therefore the processor time is maximized when ζ is at it's maximum value. This occurs when the set of fine time points calculated by the processor is located between the milestone time points of t_0 and $t = J\Delta t$. This makes sense as we need to spend time computing these fine points at all milestone points with a later time value, so the more milestone points that occur after the fine points, the longer the algorithm will take for that individual processor to complete. We can then say that the complete algorithm time is equal to the maximal individual processor time which occurs when $\zeta = 1$, the minimal value for zeta. The wall clock time can then be expressed as

$$\frac{N+\frac{N}{J}(K-1)}{K}(\gamma+\beta) + \frac{\alpha}{2}\left(\frac{N^2}{J^2}+\frac{N}{J}\right) + \left(\frac{N-\frac{N}{J}}{K}\right)\left(\frac{N}{J}-1\right)\alpha. \quad (5.5)$$

It is difficult to initially see that this is a reduction on the amount of wall clock time required for the efficient algorithm presented in [4], however we can see that there is a

reduction by an order of J in the second term. This overall reduction is easier to see when we consider the maximal case of the number of processors with $K = N - \frac{N}{J}$, under this final assumption, we can state that the minimal possible time cost for this algorithm becomes

$$\left(\frac{N}{J} + 1\right)(\gamma + \beta) + \frac{\alpha}{2}\left(\frac{N^2}{J^2} + 3\frac{N}{J} - 2\right). \tag{5.6}$$

Comparing this with equation (5.4) shows that in the limiting case, we can reduce the computational wall clock time from Chen and Wan's algorithm by a factor of approximately J, meaning that the larger time steps we take, the more savings we will see at the expense of some amount of accuracy. We see also that the computational time is still quadratic with respect to N, this could theoretically be circumvented by careful selection of J, i.e. $J = \sqrt{N}$. However, this causes J to increase with N, and thus we leave an in depth analysis of the consequences of such a choice to some future work.

We conclude this section with Figure 5.1 which shows curves of the expected computational times constructed by equations (5.3) and (5.5). Unless otherwise noted, the parameter values are as follows: $\alpha = 1.4$, $\beta = 5.6$, $\gamma = 0.1$, $N = 100$, $J = 4$, and $K = 75$. We can immediately see by comparing the blue and red curves that the expected computational time is theoretically scaled down significantly by implementing the parallel method with $K = 75$ processors. We also see that the expected time has a similar sensitivity to varying J and K, with similarly shaped curves, but at very different scales since our base case has $4 = J \ll K = 75$. These results are further numerically evidenced near the end of Chapter 6.

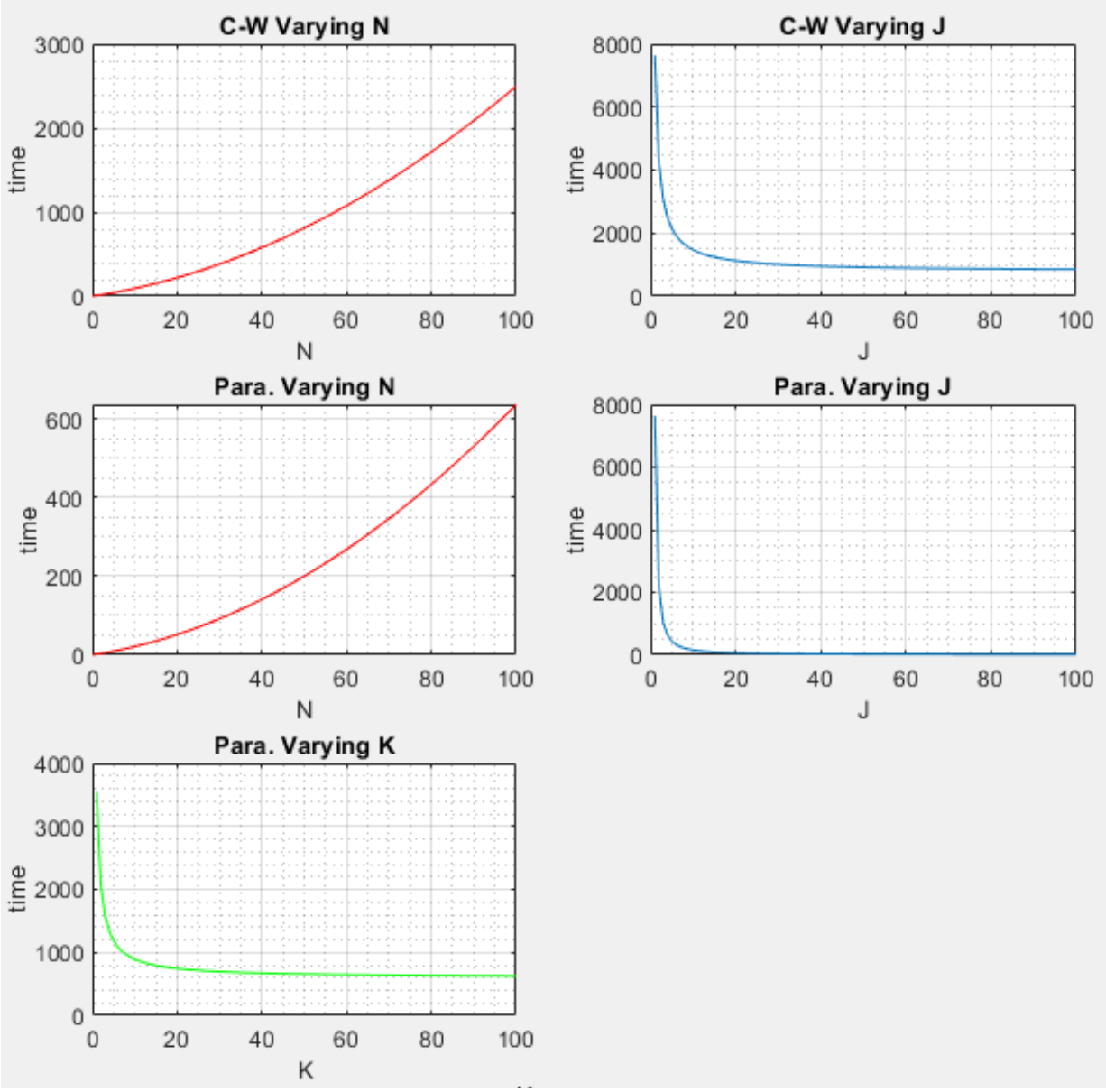

Figure 5.1: Expected computation times of Chen and Wan's algorithm varying N, J, and the proposed parallel algorithm varying N, J, K

Chapter 6

Numerical Observations

In this chapter we solve the American option pricing problem using the parallel algorithm as defined in Algorithm 3. We calculate and compare the initial values of the option $v(\vec{s}^{\,0}, 0)$ and the initial option deltas $\vec{\nabla} v(\vec{s}^{\,0}, 0)$ with the same values calculated by Chen and Wan's efficient method described in Algorithm 2. We calculate these values using $t_0 = 0$ given an initial price vector $\vec{s}^{\,0} = (s_1^0, s_2^0, \cdots, s_d^0)$ for values of $d = 7, 13, 20$. We also compute these values and deltas across the space time domain and compare them with the baseline given by the efficient methods.

Throughout all of our simulations, we consider a geometric average option with payoff value $f(\vec{s}) = \max\left(\left(\prod_{i=1}^{d} s_i\right)^{\frac{1}{d}} - X, 0\right)$ and strike prices $X = 90, 100, 110$. While these options are not the most common for practical applications, they are useful as a baseline tool to test the accuracy of the method because we can derive semi-analytical solutions to the value and deltas of the option to construct error measures for the values we compute. Glasserman [11] and Sirignano and Spiliopoulos [20] show that any d-dimensional American geometric call option of this form can be reduced to a one dimensional American call option in the variable $s' = \left(\prod_{i=1}^{d} s_i\right)^{\frac{1}{d}}$ with effective volatility $\sigma' = \sqrt{\frac{1+(d-1)\rho}{d}}\sigma$ and effective drift $r' = r - \delta + \frac{1}{2}(\sigma'^{\,2} - \sigma^2)$. Once this one dimensional option is constructed, its values can be computed via finite difference and the deltas can be computed wherever $s_1 = s_2 = \cdots = s_d$. In all experiments in this chapter, we assume a d-dimensional American geometric call option with parameter values $\rho_{i,j} = 0.75$, $\sigma = 0.25$, $r = 0$, $\delta = 0.02$, $T = 2$.

In the experiments, we compute for different values of K, N, J as described in each of the corresponding tables and we consider the maximum number of processors available for the method $N - \frac{N}{J}$. Each of the neural networks consist of $L = 7$ hidden layers of

equal width $d^{[l]} = d + 5, l = 1, \cdots, L$. Each simulation considers $M = 240000$ simulated price paths beginning at $\vec{s}^{\,0}$, and we train each network over 600 batches each of size 400. Each of the algorithms are completed on $N - \frac{N}{J}$ **Compute Canada Graham Cluster base-GPU nodes** each consisting of a single P100 Pascal GPU and 2 Intel E5-2683 v4 Broadwell @ 2.1GHz CPUs with 1024MB of memory availability.

Note that there are other parameter selections with respect to the algorithm improvements discussed in Chen and Wan [4]. We follow their suggestions and select function smoothing parameter $\kappa = \frac{2}{\Delta t}$, valuation weight $\theta = 0.5$, and we only perform a single network ensemble with $C = 1$.

6.1 Accuracy

In this section, we analyze the computational accuracy of the proposed parallel method and compare it to Chen and Wan's efficient algorithm. We propose that in most cases, the best value for considering the accuracy of the method will be to consider the initial value and deltas of the option $v(\vec{s}^{\,0}, t_0)$. We select this because in order to compute this value, the entire course network must be traversed, meaning that any inaccuracies across any of the course networks will compound through all of the computations. Thus result in the value at $t = 0$ being either the most inaccurate, or at least the most volatile since it's possible some of these inaccuracies can cancel out. Since we have the ability to compute the semi-analytical 'true' price of the option as described previously, we need to consider what error measure we will use to measure the accuracy of our proposed method. Luckily, Chen and Wan describe a suitable error measure in [4] and we continue to use this method here. We measure the absolute percent error of the computed values with respect to the finite difference solution, i.e. denote the finite difference solution v^* then

$$\frac{\|v(\vec{s}^{\,0}, 0) - v^*(\vec{s}^{\,0}, 0)\|}{\|v^*(\vec{s}^{\,0}, 0)\|} \times 100\%, \qquad \frac{\|\vec{\nabla} v(\vec{s}^{\,0}, 0) - \vec{\nabla} v^*(\vec{s}^{\,0}, 0)\|_{L_2}}{\|\vec{\nabla} v^*(\vec{s}^{\,0}, 0)\|_{L_2}} \times 100\%, \tag{6.1}$$

define the percentage error of the calculated initial option value and deltas respectively, with respect to the true value.

Tables 6.1-6.6 show the results of computing the initial values and deltas of the American geometric option via the proposed parallel algorithm and Chen and Wan's efficient algorithm, as well as their percentage error with respect to the semi-analytical true value of the option. For all of these simulations, we keep constant the number of total time steps

$N = 100$, as well as the milestone step size $J = 4$. From this we also take the maximal amount of reasonable processors $K = N - \frac{N}{J} = 75$, although discussion and results on the computational wall clock time for each simulation is shown later, we also include in parentheses the amount of time (in seconds) that the algorithm took to finish.

Table 6.1: Geometric Option Initial Values with dimensionality = 7

s_i^0	Exact Price $v(\vec{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Value	% Error	Computed Value	% Error
90	5.9021	5.8882 (614sec)	0.24%	5.8822 (2531sec)	0.34%
100	10.2591	10.1928 (620sec)	0.65%	10.2286 (2472sec)	0.30%
110	15.9878	15.9813 (611sec)	0.04%	15.9738 (2491sec)	0.09%

Table 6.2: Geometric Option Initial Values with dimensionality = 13

s_i^0	Exact Price $v(\vec{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Value	% Error	Computed Value	% Error
90	5.7684	5.7310 (984sec)	0.65%	5.7719 (4150sec)	0.06%
100	10.0984	10.0392 (1020sec)	0.59%	10.1148 (4213sec)	0.16%
110	15.8200	15.7926 (944sec)	0.17%	15.8259 (4087sec)	0.04%

Table 6.3: Geometric Option Initial Values with dimensionality = 20

s_i^0	Exact Price $v(\vec{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Value	% Error	Computed Value	% Error
90	5.7137	5.6811 (4749sec)	0.57%	5.7105 (22043sec)	0.06%
100	10.0326	9.9967 (4578sec)	0.36%	10.0180 (20626sec)	0.15%
110	15.7513	15.7057 (4884sec)	0.29%	15.7425 (21369sec)	0.06%

We can see from the observations shown in Tables 6.1-6.3 that the proposed algorithm continues to show a high degree of accuracy when computing the initial option values. The majority of computed values lie within 0.5% of the true value and none are worse than 0.65%. We see that for all three values of d, the best computed values are when $K = 110 > s_i^0$, it's difficult to state this is a trend with a small sample size however and this is likely just due to statistical variance. We also see that in terms of accuracy

our method does slightly worse overall than Chen and Wan's method, which is difficult to explain since theoretically we should get the same or highly similar outputs from both methods. However, one possible explanation, which is also discussed later is it's possible we see higher levels of variation since each processor independently simulates it's own price paths over larger time steps, instead of constructing a full domain of price paths and distributing the necessary trajectories to each processor.

Table 6.4: Geometric Option Initial Deltas with dimensionality = 7

s_i^0	Exact Deltas $\vec{\nabla} v(\bar{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Deltas	% Error	Computed Deltas	% Error
90	$(0.0523, \cdots, 0.0523)$	$(0.0525, \cdots, 0.0525)$	0.38%	$(0.0516, \cdots, 0.0516)$	1.2%
100	$(0.0722, \cdots, 0.0722)$	$(0.0737, \cdots, 0.0737)$	2.08%	$(0.0710, \cdots, 0.0710)$	1.7%
110	$(0.0912, \cdots, 0.0912)$	$(0.0909, \cdots, 0.0909)$	0.33%	$(0.0901, \cdots, 0.0901)$	1.2%

Table 6.5: Geometric Option Initial Deltas with dimensionality = 13

s_i^0	Exact Deltas $\vec{\nabla} v(\bar{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Deltas	% Error	Computed Deltas	% Error
90	$(0.0279, \cdots, 0.0279)$	$(0.0279, \cdots, 0.0279)$	$< 0.40\%$	$(0.0277, \cdots, 0.0277)$	0.76%
100	$(0.0387, \cdots, 0.0387)$	$(0.0385, \cdots, 0.0385)$	0.52%	$(0.0384, \cdots, 0.0384)$	0.83%
110	$(0.0492, \cdots, 0.0492)$	$(0.0488, \cdots, 0.0488)$	0.81%	$(0.0486, \cdots, 0.0486)$	1.1%

Table 6.6: Geometric Option Initial Deltas with dimensionality = 20

s_i^0	Exact Deltas $\vec{\nabla} v(\bar{s}^0, 0)$	Parallel Algorithm		Chen-Wan Algorithm	
		Computed Deltas	% Error	Computed Deltas	% Error
90	$(0.0180, \cdots, 0.0180)$	$(0.0180, \cdots, 0.0180)$	$< 0.55\%$	$(0.0179, \cdots, 0.0179)$	0.7%
100	$(0.0251, \cdots, 0.0251)$	$(0.0250, \cdots, 0.0250)$	0.40%	$(0.0248, \cdots, 0.0248)$	1.2%
110	$(0.0320, \cdots, 0.0320)$	$(0.0318, \cdots, 0.0318)$	0.63%	$(0.0316, \cdots, 0.0316)$	1.2%

Tables 6.4-6.6 present the computed initial deltas under the same simulation parameters that were used for the previous tables, as well as the percentage errors of the proposed

method and Chen and Wan's method with respect to the true value. We notice an interesting phenomenon occurring here, while the initial option prices were generally computed slightly worse by the parallel algorithm, the computed deltas are all more accurate than Chen and Wan's method, except in one case. Again, there's no theoretical reason we can construct as to why this has occurred other than standard statistical variance, however it's possible that along the coarser grid with fewer networks we avoid overfitting a little bit better in this case, resulting in more accurate solutions. This explanation however does not explain the less accurate option prices and more in depth analysis into this phenomenon could be a focus of future research works.

Now that we have constructed a number of experiments which illustrate the numerical accuracy of the proposed method, we turn our attention to another small subproblem of interest. We are interested in the effect of changing values of N and J on the overall accuracy of the method, intuition says that taking larger step sizes on either the coarse or fine grid should lead to less accurate results. We simulate a number of experiments to test this idea and the results are compiled in Table 6.7. For these experiments we set $K = s_i^0 = 100$, and all other constants are the same as previous experiments

Table 6.7: Geometric Option Initial Value with dimensionality = 7

Exact Price $v(\vec{s}^0, 0)$		Parallel Algorithm			Percentage Error		
		N=100	N=60	N=20	N=100	N=60	N=20
10.2591	J=4	10.1928	10.2581	10.2386	0.65%	0.01%	0.20%
	J=5	10.2680	10.2519	10.2220	0.09%	0.07%	0.36%
	J=10	10.2579	10.1904	10.1838	0.01%	0.67%	0.73%

We notice interestingly enough that for the simulations with $N = 100$, the accuracy increases with J, in defiance with the assumed relationship discussed previously. However, for all other values of N, our intuitive theory is supported as the accuracy diminishes with larger coarse grid step sizes. We can chalk the increase in accuracy for $N = 100$ due to statistical variance with a low sample size as no theoretical reason could be constructed for this relationship.

Finally, after computing the simulations and discussing the accuracy results so far, we finish by presenting one of the completed algorithms in it's completed form. Figure 6.1 shows the outputs of Chen and Wan's efficient algorithm (left) and the proposed parallel algorithm (right). The figure graphs the geometric average of the underlying assets against time, with each dot on the graph representing one price vector at a certain point in time. All 240000 price trajectories are shown and the colours represent the execution barrier

with blue representing a price vector where the option should be exercised since the value is currently larger than the future expected value, and red dots represent continuation price vectors. We notice two aspects of the graphs that are worth discussing, firstly, we see that the networks in the proposed parallel algorithm successfully learn the exercise dynamics of the problem, reproducing the output of Chen and Wan's method. Secondly, we notice that the distribution of points in the parallel algorithm is not as smooth as the previous method. This, as discussed before, is due to the independent price path generation between the individual processors, and this could be fixed by having one processor generate the full fine grid of Monte Carlo paths and then distributing the necessary points to the individual processors. A second solution is to have each individual processor generate the paths across N points with the same global seed, and only save the points that it needs to calculate.

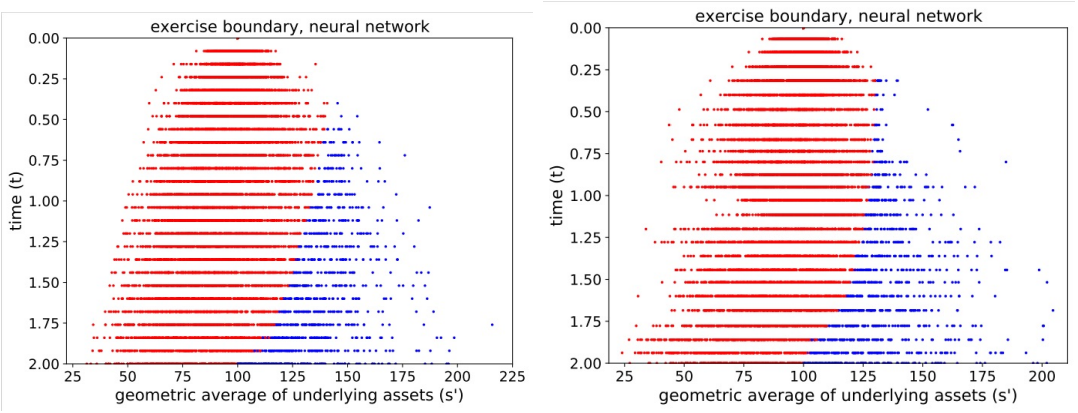

Figure 6.1: Solved exercise dynamics by Chen and Wan's efficient algorithm (left) and the proposed parallel algorithm (right)

6.2 Computational Time

The motivation behind the proposed parallel algorithm was to reduce the computational time required to solve the American option pricing problem, and thus allow for more attainable market information amongst investors. As was done in the previous section, we will analyze the required wall clock time to complete the proposed parallel algorithm, as well as Chen and Wan's efficient method. By doing this we can construct the clock time reduction ratio to see how changing parameters affect the proportional amount of wall clock time saved. Denote W_p and W_c as the clock times to complete the parallel and

Chen-Wan algorithm respectively, then the wall clock reduction ratio, R, is simply $R = \frac{W_c}{W_p}$. The parameters for the simulations in Table 6.8 are equivalent to what was used for the results of table 6.7.

Table 6.8: Geometric Option Computational Time with dimensionality = 7

	Parallel Algorithm			Chen-Wan Algorithm			Clock Time Reduction Ratio		
	N=100	N=60	N=20	N=100	N=60	N=20	N=100	N=60	N=20
J=4	689	251	61	2472	1102	215	3.59	4.39	3.52
J=5	419	178	54	2136	941	209	5.10	5.29	3.87
J=10	151	71	32	1491	721	183	9.87	10.15	5.72

First we quickly point out that for the experiment in Table 6.8 with $N = 100$ and $J = 4$, we noted that approximately the average time to complete a Monte Carlo price path step was $\gamma \approx 0.1$ seconds. Similarly the amount of time to train each neural network as well as update the values of the option at all prices within a specific time step as $\beta \approx 5.6$ and $\alpha \approx 1.4$ seconds respectively. We can then numerically provide evidence for our theoretical time requirements by plugging these values into equations (5.3) and (5.6). Doing so gives us an expected clock time of $100(0.1 + 5.6) + \frac{1.4}{2}(\frac{100^2}{4} + 3(100) - 2\frac{100}{4}) = 2495$ seconds for Chen and Wan's method and $(\frac{100}{4} + 1)(0.1 + 5.6) + \frac{1.4}{2}(\frac{100^2}{4^2} + 3\frac{100}{4} - 2) = 636.8$ seconds for the proposed parallel algorithm. The numerical results of 2472 and 689 seconds for each respective method provide numerical evidence to back up the theoretical equations, the discrepancy is likely due to an sampling error in approximating the average time for each process in the algorithm as it's not exactly constant between any two simulations.

We also note that in Chapter 5 it was suggested that the reduction of Chen and Wan's algorithm was approximately equal to J, and we see numerical evidence to support this in Table 6.8 with the reduction ratio being around J for both of the two larger values of N. However, we also see that when N is restricted to be relatively small with respect to the problem and J constitutes a large portion of N, i.e. $N = 20, J = 10$ we see that the reduction ratio is smaller than expected. Which could be due to the fact that the algorithm is completed fast enough regardless, that the clock time required to complete background processes not considered in the theoretical analysis, i.e. memory allocation, information passing between processors, etc. constitute a larger portion of the wall clock time, resulting in a lower reduction ratio. In the smaller discretizations, these background processes are likely dominated by the algorithm itself and thus we see closer ratios to what is expected theoretically.

Chapter 7

Conclusion

We propose a parallel processing implementation of a neural network architecture developed into a previous American option pricing algorithm. The parallel algorithm uses two levels of discrete grid in order to partition the work of the algorithm onto separate processors, resulting in large computational time and memory cost saves. The proposed algorithm not only saves plenty of computational time, but also maintains the order of accuracy of previous methods both in computing option prices and deltas. We numerically and theoretically explore the computational time saves of completing the parallel algorithm over the methods constructed in previous works and show evidence for the usefulness of the parallel method to market investors.

We recognize a number of drawbacks to the current implementation of the proposed parallel method, including that the computational time cost increases quadratically with the size of the discretization of the spatial domain. This could theoretically be circumvented by careful selection of the size of the coarse grid with respect to the finer grid, however this is an avenue of research for some future work. We also recognize that the selection to have each processor independently construct it's own Monte Carlo paths leads to a less smooth distribution of price vectors in the final result. This could be fixed in a future implementation of the algorithm which could reduce the level of variation in the solution of the problem. A future area of work into establishing optimal parameter values could prove essential into pushing the solutions of American option prices further, allowing us to price further into the future and more often, resulting in more accurate delta hedging processes and limiting exposure to market risk.

References

[1] Y. Achdou and O. Pironneau. *Computational methods for option pricing.* Frontiers in Applied Mathematics Vol. 30, 2005.

[2] G. Bal and Y. Maday. *A 'Parareal' Time Discretization for Non-Linear PDE's with Application to the Pricing of an American Put.* Recent Developments in Domain Decomposition Methods, 2002.

[3] E W. Beck, C. and A. Jentzen. *Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations.* Journal of Nonlinear Science, pp. 1-57, 2017.

[4] Yangang Chen and Justin W. L. Wan. *Deep Neural Network Framework Based on Backward Stochastic Differential Equations for Pricing and Hedging American Options in High Dimensions.* Quantitative Finance, Vol. 21, Iss. 1, 2019.

[5] D.J. Duffy. *Finite difference methods in financial engineering.* Wiley Finance Series, John Wiley Sons, Ltd., Chichester, A partial differential equation approach, With 1 CD-ROM (Windows, Macintosh and UNIX)., 2006.

[6] Han J. E, W. and A. Jentzen. *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations.* Commun. Math. Stat., Vol. 5, pp. 349-380, 2017.

[7] Peng S. El Karoui, N. and M.C. Quenez. *Backward stochastic differential equations in finance.* Math. Finance, Vol. 7, pp. 1–71, 1997.

[8] Friedhoff S. Kolev Tz. V. MacLachlan S. P. Falgout, R. D. and J.B. Schroder. *Parallel Time Integration with Multigrid.* SIAM Journal on Scientific Computing, Vol. 36, Iss. 6, pp. 635-661, 2014.

[9] P.A. Forsyth and K.R. Vetzal. *Quadratic convergence for valuing American options using a penalty method*. SIAM J. Sci. Comput., Iss. 23, pp. 2095–2122, 2002.

[10] Takahashi A. Fujii, M. and M. Takahashi. *Asymptotic Expansion as Prior Knowledge in Deep Learning Method for high dimensional BSDEs*. arXiv preprint arXiv:1710.07030, 2017.

[11] P. Glasserman. *Monte Carlo methods in financial engineering*. Applications of Mathematics (New York) Vol. 53, Springer-Verlag, New York, Stochastic Modelling and Applied Probability, 2004.

[12] Bengio Y. Goodfellow, I. and A. Courville. *Deep Learning, Adaptive Computation and Machine Learning*. MIT Press, Cambridge, MA., 2016.

[13] Jentzen A. Han, J. and W. E. *Solving high-dimensional partial differential equations using deep learning*. Proc. Natl. Acad. Sci. USA, Iss. 115, pp. 8505-8510, 2018.

[14] J.C. Hull. *Options futures and other derivatives*. Pearson/Prentice Hall, 2003.

[15] D.P. Kingma and J. Ba. *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980, 2014.

[16] M Kohler. *A review on regression-based Monte Carlo methods for pricing American options*. Recent developments in applied probability and statistics, pp. 37–58, Springer, 2010.

[17] C.C.W. Leentvaar. *Pricing multi-asset options with sparse grids*. 2008.

[18] F.A. Longstaff and E.S. Schwartz. *Valuing American options by simulation: a simple least-squares approach*. The review of financial studies, 14, 113–147, 2001.

[19] C. Reisinger and J.H. Witte. *On the use of policy iteration as an easy way of pricing American options*. SIAM J. Financial Math., Iss. 3, pp. 440-458, 2012.

[20] J. Sirignano and K. Spiliopoulos. *DGM: A deep learning algorithm for solving partial differential equations*. J. Comput. Phys., Iss. 375, pp. 1339-1364, 2018.

[21] L. Stentoft. *Convergence of the least squares Monte Carlo approach to American option valuation*. Management Science, Vol. 50, pp. 1193–1203, 2004.

[22] J.N. Tsitsiklis and B. Van Roy. *Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives*. IEEE Trans. Automat. Control, Iss. 44, pp. 1840-1851, 1999.

[23] Lai K. Kolkiewicz A. W. Wan, J. W. L. and K. S. Tan. *A Parallel Quasi-Monte Carlo Approach to Pricing American Options on Multiple Assets*. International Journal of High Performance Computing and Networking, Vol. 4, Iss. 5, pp. 321-330, 2006.