Mathematical Finance MSc Dissertation MTH775P, 2018/19

Accelerated Grids

Optimizing Solvers for Financial Partial Differential Equations

Mustafa Berke Erdis, ID 180883925

Supervisor: Dr. Sebastian del Baño Rollin

A thesis presented for the degree of
Master in Sciences in *Mathematical Finance*

School of Mathematical Sciences
and *School of Economics and Finance*
Queen Mary University of London

Declaration of original work

This declaration is made on September 3, 2019.

Student's Declaration: I, Mustafa Berke Erdis, hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text, nor has any part been written for me by another person.

Referenced text has been flagged by:

1. Using italic fonts, **and**
2. using quotation marks "...", **and**
3. explicitly mentioning the source in the text.

Acknowledgements

I would like to thank my supervisor Dr Sebastian del Baño Rollin for the encouragement and guidance through the independent learning process of this master thesis. I must express my very profound gratitude to my parents for providing me with endless support throughout my years of education. This accomplishment would not have been possible without them.

Abstract

Derivative pricing plays a critical role in the day to day operations of modern financial corporations. Accurate pricing of derivatives in the real world is a computationally intensive task consisting of solving partial differential equations. Considering that most of the time determining an analytical solution is impossible, numerical methods used to solve partial differential equations. This study aims to determine how financial institutions can optimize numerical methods using available resources. To answer this question, we compared the performance of different solution platforms, compilers, optimization switches and tridiagonal system solvers. Against the base case using x86 platform, Visual Studio compiler, Thomas algorithm and no optimization flags, the results indicate that utilizing the Intel compiler, parallelized cyclic reduction and /Ox optimization flag improved the timings by 35.5% and 30.5% for the heat equation and Black - Scholes equation. Further research is needed to identify other factors that could improve the timings of these numerical methods.

Queen Mary University of London
3rd September 2019

Contents

Chapter 1

Introduction

In Ancient Greece, Thales was scorned for his poverty. Later that year, Thales utilized his skills in astrology to forecast an increase in olive yields. Using his limited capital, he rented oil presses in winter. Months later, over the oil making season, many people rushed to the presses because of the high yields that Thales predicted. As he rented the presses over the winter, he forced the terms he pleased. Thales showed it was easy for philosophers to be rich if they chose it and practically used the first financial derivative product [1].

In the modern world, financial derivatives are contracts between two or more parties. The value of the contract depends on one or several underlying assets. Commonly the assets are currencies, equities, bonds, interest rates, market indices or commodities. The vanilla call option gives the right but not the obligation to buy the underlying asset at the expiry date at a previously agreed strike price. Essentially, Thales bought call options for oil presses. If the olive yields didn't come as Thales expected he didn't have the obligation to use the olive presses. On the other hand, the vanilla put option gives the right but not the obligation to sell the underlying asset at the expiry date at a previously agreed strike price. Practical applications of the options include hedging or speculating the future asset price. Hence, accurately pricing the options is crucial for an efficient and mature financial market.

Black - Scholes model was introduced by Fischer Black and Myron Scholes to fix the option pricing problem in 1973 [2]. Merton extended the framework to allow jumps in the underlying assets [22]. Besides option valuation, the framework gives a new perspective of hedging using the derivatives of the option price, the Greeks. The formula is used by thousands of traders and investors on the day to day tasks to value options and manage risk throughout the world financial markets. This new method of determining the value of derivatives by Merton and Scholes received the Nobel Prize in Economic Science in 1997 [33]. The Black - Scholes model is a parabolic partial differential equation that the price of option satisfies under certain conditions and assumptions. Determining an analytical solution to partial differential equations are impossible in most cases. Therefore, using numerical methods are essential. They were first applied in mathematical finance using the explicit method in 1977 for option valuation by Brennan and Schwartz [3].

1.1 Motivation of the Project

Derivative pricing in the real world is a computationally intensive task. The existing numerical methods for partial differential equations are all constrained by the computational complexity. Being fast when evaluating new information is critical for the operations of financial institutions. Hence, optimizing the existing numerical methods with hardware and software that can be installed on a trading floor is crucial. Purpose of this project to achieve faster numerical solutions for financial partial differential equations with the given resources.

The existing explicit scheme, Crank - Nicolson (CN) method and alternating direction implicit (ADI) scheme is used to find solutions to partial differential equations. Firstly, Visual Studio compiler for C++ is replaced by Intel compiler to explore the effects on the speed of the solver. Following the compiler tests, x86 and x64 solution platforms are used to test the compilers against each other. The penultimate test was to study the effects of Visual Studio optimization switches to achieve a faster option pricing. Finally, an important consideration in this thesis is the efficient solution of the tridiagonal systems arising at each time step when using CN and ADI schemes. In this case, the standard tridiagonal system solvers, such as the Thomas algorithm or Gaussian elimination, may not be an efficient choice. To take advantage of the high-performance CPU parallelism techniques, the cyclic reduction method combined with OpenMP is considered.

Chapter 2

Pricing Financial Derivatives

2.1 The Risk Neutral Approach

The Black-Scholes framework is a theoretical valuation formula for options. It reveals the relationship between the prices of the options and the underlying assets. Since almost all corporate liabilities can be viewed as combinations of options, the formula can be applied to common stocks and corporate bonds [2]. The Black-Scholes model makes the following assumptions:

- There does not exist any arbitrage opportunity in the financial market. The traders can not make profits without any risk.
- The underlying asset value follows a geometric Brownian Motion, $\mathrm{d}S = \mu S \mathrm{d}t + \sigma S \mathrm{d}B$ where μ, denotes the average rate of growth of the underlying assets, σ denotes the volatility of the asset price and B is a Brownian Motion.
- The market is frictionless. This means there are no transaction fees, the interest rates for borrowing and lending money from and to the bank are the same, every party in the market has immediate information and all entities are available at any time and in any size.

2.1.1 Black-Scholes Partial Differential Equation

The original Black-Scholes model is used to price vanilla options. Presence of dividends can be included in the Black-Scholes formula. Since it doesn't affect the performance, for the sake of simplicity we will assume there are no dividends paid. Under the assumptions of the Black-Scholes framework, the call or put option price satisfies the following parabolic partial differential equation.

$$\frac{\partial V}{\partial t} = rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rV \tag{2.1}$$

The framework denotes the price $V(t, S)$ of a European option, which is driven by the underlying asset that satisfies the PDE. r, σ, t, S respectively denote the risk-free interest rate, volatility, time and the underlying price. It is assumed that r and σ are constants. In more complicated models such as stochastic volatility, they are modelled as a function. We will consider the PDE and conditions for the European call options. In order to price a vanilla call option, the PDE needs to satisfy the following boundary and initial conditions.

$$C(0, t) = 0, \qquad 0 \le t \le T \tag{2.2}$$

$$C(S_{\max}, t) = S_{\max} - Ke^{-r(T-t)}, \qquad 0 \le t \le T \tag{2.3}$$

$$C(S, T) = \max(S - K, 0), \qquad 0 \le S \le S_{\max} \tag{2.4}$$

2.1.2 Derivation of the Black-Scholes Equation

Firstly, we will introduce the properties of the geometric Brownian motion and Itô's lemma to derive the Black-Scholes equation.

Definition 2.1.1. Brownian motion (also known as Wiener Process) was discovered by botanist Robert Brown as he observed a chaotic motion of particles suspended in water [38]. A Brownian motion, $B(t)$, is a continuous-time stochastic process with the following properties:

- $B(0) = 0$.
- $B(t)$ is a continuous function of t.
- For $0 \le s < t$ the increment $B(t) - B(s)$ has normal distribution $\mathcal{N}(0, t - s)$.
- For $t_0 \le t_1 \le ... \le t_n$ the increments $B(t_k) - B(t_{k-1})$ where $k = 1, ..., n$ are independent random variables.

Brownian motion is the basic building block in stochastic calculus and geometric Brownian motion is used to model the stock prices in the Black-Scholes model.

Lemma 2.1.2. *Itô's Lemma: Let $B(t)$ be a Brownian motion and $X(t)$ be an Ito process which satisfies the stochastic differential equation:*

$$\mathrm{d}X(t) = \mu(X(t),t)\mathrm{d}t + \sigma(X(t),t)\mathrm{d}B(t) \tag{2.5}$$

If $f(x,t)$ is twice continuously differentiable function then $f(X(t),t)$ is also an Itô drift-diffusion process [18], with its differential given by:

$$\mathrm{d}(f(X(t),t)) = \frac{\partial f}{\partial t}(X(t),t)\mathrm{d}t + f'(X(t),t)\mathrm{d}X + \frac{1}{2}f''(X(t),t)\mathrm{d}X(t)^2 \tag{2.6}$$

With $\mathrm{d}X(t)^2$ given by: $\mathrm{d}t^2 = 0$, $\mathrm{d}t\mathrm{d}B(t) = 0$ and $\mathrm{d}B(t)^2 = \mathrm{d}t$.

Theorem 2.1.3. *Assume that the asset price S follows a geometric Brownian motion. Under the assumptions of Black-Scholes framework, the call or put option price $V(t,S)$ satisfies the parabolic partial differential equation*

$$\frac{\partial V}{\partial t} = rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rV \tag{2.7}$$

Proof. Suppose an investor sets up a self-financing portfolio, $X(t)$, comprising one option and an Δ amount of the underlying asset. Therefore, value of the portfolio at time t is $X(t) = V(t) + \Delta S(t)$. Since the self-financing trading strategy has no capital influx or consumption, the value of portfolio change can be written as

$$\mathrm{d}X = \mathrm{d}V + \Delta\mathrm{d}S \tag{2.8}$$

Applying the Itô's Lemma to the option price V(t,S)

$$\mathrm{d}V = \frac{\partial V}{\partial t}\mathrm{d}t + \frac{\partial V}{\partial S}(S,t)\mathrm{d}S + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}(S,t)\mathrm{d}S^2 \tag{2.9}$$

Since the Black-Scholes model assumes that the stock price under the "market probability measure" follows a gBM.

$$\mathrm{d}S = \mu S\mathrm{d}t + \sigma S\mathrm{d}W \tag{2.10}$$

Putting 2.10 and 2.9 together yields

$$\mathrm{d}V = (\frac{\partial V}{\partial t} + \mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \Delta\mu S)\mathrm{d}t + (\sigma S\frac{\partial V}{\partial S} + \Delta\sigma S)\mathrm{d}W \tag{2.11}$$

The fact that portfolio is risk-free implies that the second term involving the Brownian Motion, $\mathrm{d}W$, must be zero. This technique is known as delta-hedging , otherwise, we would have an arbitrage opportunity. Thus, $\Delta = -\frac{\partial V}{\partial S}$. Hence, the growth rate of the portfolio must be the risk free rate which can be summarized as $\mathrm{d}X = rX\mathrm{d}t$. Substituting Δ and $\mathrm{d}X$ yields

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = r(V - S\frac{\partial V}{\partial S}) \tag{2.12}$$

Rearranging the equation to get famous Black-Scholes equation:

$$\frac{\partial V}{\partial t} = rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rV \tag{2.13}$$

□

Definition 2.1.4. The resulting partial differential equation can be solved analytically using the following boundary conditions and initial conditions for call options.

$$\begin{aligned} \frac{\partial C}{\partial t} &= rS\frac{\partial C}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} - rC && (2.14)\\ C(0,t) &= 0,\ C(S_{\max},t) = S_{\max} - Ke^{-r(T-t)}, \quad 0 \le t \le T && (2.15)\\ C(S,T) &= \max(S-K,0), \quad 0 \le S \le S_{\max} && (2.16) \end{aligned}$$

Theorem 2.1.5. *In order to solve the Black-Scholes equation analytically we need the Feynman - Kac Theorem [18]. Suppose that x_t follows the process*

$$\mathrm{d}x_t = \mu(x_t,t)\mathrm{d}t + \sigma(x_t,t)\mathrm{d}W_t^Q \tag{2.17}$$

.

Assume that there is a solution to the function $V(x_t,t)$ that follow the partial equation

$$\frac{\partial V}{\partial t} + \mu(x_t,t)\frac{\partial V}{\partial x} + \frac{1}{2}\sigma(x_t,t)^2\frac{\partial^2 V}{\partial x^2} - r(t,x)V(x_t,t) = 0 \tag{2.18}$$

.

The solution to the function under the measure Q is

$$V(x_t,t) = E^Q[\exp(-\int_t^T r(X_u,u)\mathrm{d}u)V(X_T,T)|\mathcal{F}_t] \tag{2.19}$$

Theorem 2.1.6. *Solving the equations, the formulae [48] for European call is*

$$\begin{aligned} C &= S\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2) && (2.20) \\ d_1 &= \frac{\log(S/K) + (r+\sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} && (2.21) \\ d_2 &= d_1 - \sigma\sqrt{T-t} && (2.22) \end{aligned}$$

where $\Phi(x)$ denotes the cumulative normal distribution function.

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-\frac{t^2}{2})\mathrm{d}t \tag{2.23}$$

Proof. Applying the Feynman - Kac formula 2.1.5 to the Black-Scholes equation with boundary conditions yields

$$\begin{aligned} C(S_t, t) &= E^Q[e^{-\int_t^T r(X_u,u}\mathrm{d}u)C(S_T, T)|\mathcal{F}_t] && (2.24) \\ &= e^{-r(T-t)}E^Q[(S_T - K)^+|\mathcal{F}_t] && (2.25) \end{aligned}$$

Evaluating the expectation as integral

$$\begin{aligned} C(S_t, t) &= e^{-r(T-t)} \int_K^\infty (S_T - K)\mathrm{d}F(S_t) && (2.26) \\ &= e^{-r(T-t)} \int_K^\infty S_T\mathrm{d}F(S_t) - e^{-r(T-t)} \int_K^\infty K\mathrm{d}F(S_t) && (2.27) \end{aligned}$$

As the stock price S follow log-normal distribution [15] the first integral becomes

$$\begin{aligned} \int_K^\infty S_T\mathrm{d}F(S_t) &= E^Q[S_T|S_T > K] && (2.28) \\ &= S_t e^{r(T-t)}\Phi(\frac{\log(S/K) + (r+\sigma^2/2)(T-t)}{\sigma\sqrt{T-t}}) && (2.29) \\ &= S_t e^{r(T-t)}\Phi(d_1) && (2.30) \end{aligned}$$

Thus, second integral in 2.26 can be written as

$$\begin{aligned} e^{-r(T-t)}K\int_K^\infty \mathrm{d}F(S_t) &= e^{-r(T-t)}K(1-F(K)) && (2.31)\\ &= e^{-r(T-t)}K(1-\Phi(\frac{\ln(K/S_t)-(r-\frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}})) && (2.32)\\ &= e^{-r(T-t)}K(1-\Phi(\sigma\sqrt{T-t}-d_1) && (2.33)\\ &= e^{-r(T-t)}K\Phi(d_2) && (2.34)\end{aligned}$$

Combining both integrals yield the analytical solution for the Black-Scholes equation with the given boundary conditions.

$$C = S\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2) \tag{2.35}$$

□

The following parameters for Black-Scholes equation will be used for the research purposes of this project.

Parameter	Value
Strike Price (K)	1.0
Volatility (σ)	20 %
Risk Free Rate (r)	5 %
Time to Expiry (T)	2.0
Maximum Share Price (S_{max})	2.0

Definition 2.1.7. Black-Scholes PDE has coefficients that depend on the underlying price, S. Meaning that PDE is not space homogeneous. Log-Spot Black-Scholes PDE is an economically intrinsic way of looking at numbers. Two assets can be similar, however their prices can be way different. Therefore, it is conventional to investigate using the $x = lnS$ conversion. First step is to apply the chain rule to the first and second order derivatives $\frac{\partial C}{\partial S}$, $\frac{\partial^2 C}{\partial S^2}$.

$$\frac{\partial C}{\partial S} = \frac{\partial C}{\partial x}\frac{\partial x}{\partial S} = \frac{\partial C}{\partial x}\frac{1}{S} \tag{2.36}$$

$$\frac{\partial^2 C}{\partial S^2} = \frac{\partial}{\partial S}(\frac{\partial C}{\partial S}) = \frac{\partial}{\partial S}(\frac{\partial C}{\partial x}\frac{1}{S}) = -\frac{1}{S^2}\frac{\partial C}{\partial x} + \frac{\partial}{\partial S}\frac{\partial C}{\partial x}\frac{1}{S} = \tag{2.37}$$

$$= -\frac{1}{S^2}\frac{\partial C}{\partial x} + \frac{\partial^2 C}{\partial x^2}\frac{1}{S^2} \tag{2.38}$$

Substituting the transformed derivatives into the original PDE

$$\frac{\partial C}{\partial t} = (r - \frac{\sigma^2}{2})\frac{\partial C}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 C}{\partial x^2} - rC \tag{2.39}$$

The transformation creates a PDE with constant coefficients rather than coefficients that depend on S.

Remark 2.1.8. Untradable Assets
Modern financial engineering created derivatives using untradable assets as an underlying such as multi asset derivatives like equity baskets, weather derivatives, non-deliverable swaps and non-deliverable forwards. Non-deliverable forwards are for offshore investors that want to trade non-convertible currencies such as Brazilian Real, South Korean Won. The Black-Scholes model is still used in these cases [17] [5] but not entirely applicable to assets that cannot be hedged.

2.2 Partial Differential Equations

Since the foundation of the world humanity tried to understand and model nature. Mathematics describes natural phenomena such as heat, sound and flow using differential equations. They can be classified into two categories. Ordinary differential equations serve to model a movement space or plane such as the trajectory of a projectile launched from a cannon. The ordinary differential equation is used to model the curve of the trajectory. On the other hand, partial differential equations models a function, a typical example is the heat distribution. This distinction usually makes PDEs much harder to determine an analytical solution than ordinary differential equations. Therefore, we need to achieve a numerical solution to the problem. One of the most common numerical methods for partial differential equations is the finite difference methods. The schemes consist of finding approximate solutions to the problem at a discrete set of points, normally on a rectangular grid of points. Finite difference methods are simple to construct and analyse but can compromise performance because of increased computational complexity when there are high dimensions.

Feynman-Kac theorem 2.1.5, establishes the link between partial differential equations and stochastic processes by writing the solution as a conditional expectation. The theorem enabled Monte Carlo method to be utilized to find the numerical solutions to the partial differential equations. The convergence rate of Monte Carlo method for n simulations can be denoted as $\mathcal{O}(n^{\frac{-1}{2}})$ which holds for all dimensions (d). The error in d dimensional trapezoidal rule for twice continuously differentiable integrands is $\mathcal{O}(n^{\frac{-2}{d}})$ [13]. Thus, Monte Carlo is a method of choice when evaluating higher dimensions. In our calculations, we will test the case where $T = 0.06$ and $x_{max} = 1.0$. The implementation of partial differential equation class used in this work can be seen under Appendix A.

2.2.1 Heat Equation

The heat equation is fundamental to financial engineering. Heat equation is a component in the Black-Schole equation and Black-Scholes equation can be transformed to the heat equation by changing variables [49]. Therefore, understanding heat equation is crucial to grasping concepts of partial differential equations. Heat equation will serve as a benchmark with the following initial and boundary conditions.

$$u_t(x,t) = u_{xx}(x,t) \tag{2.40}$$

$$u(0,t) = u(x_{max},t) = 0, \quad 0 \le t \le T \tag{2.41}$$

$$u(x,0) = \sin(\pi x), \quad 0 \le x \le x_{\max} \tag{2.42}$$

Heat equation $u(x,t)$ is a dissipative partial differential equation, describing the dissipation of heat in a region. Physically, as time progresses heat flows to cooler regions from the warmer regions. The Laplacian operator returns the difference between average value of the function around a given point, and the value at the point. The Laplacian of the heat equation, $\Delta u = u_{xx}$, governs the heat flow and tells whether the surrounding points are hotter or colder at a given point. If there is an injustice distribution of heat at a certain point such that $\Delta u < 0$ or $\Delta u > 0$, heat will flow will from hotter regions to adjacent colder regions until there is no privilege and $\Delta u = 0$. This phenomenon is also known as the tax system or robin hood principle.

Theorem 2.2.1. *Given the initial and boundary conditions, heat equation satisfies the following analytical solution.*

$$u(x,t) = \exp(-\pi^2 t)\sin(\pi x) \tag{2.43}$$

Proof. The analytical solution of heat equation is derived by a technique called separation of variables.

$$u(x,t) = X(x)T(t) \tag{2.44}$$

$$u_{xx}(x,t) = X^{''}(x)T(t) \tag{2.45}$$

$$u_t(x,t) = X(t)T^{'}(t) \tag{2.46}$$

Using the partial derivatives the equation $u_t = u_{xx}$ becomes

$$\frac{T^{'}(t)}{T(t)} = \frac{X^{''}(x)}{X(x)} \tag{2.47}$$

Right hand side only depends on x and the left hand side depends only on t. Therefore, the equation is valid only when each side is equal to a constant, which we set to λ. Rearranging terms gives us the following equations:

$$\frac{T^{'}(t)}{T(t)} = \frac{X^{''}(x)}{X(x)} = -\lambda \tag{2.48}$$

$$X^{''}(x) + \lambda X(x) = 0 \tag{2.49}$$

$$T^{'}(t) + \lambda T(t) = 0 \tag{2.50}$$

$$X(0) = X(1) = 0 \tag{2.51}$$

Solving for X(x) is an example case of Sturm-Liouville problem [21] with three cases.

- Let $\lambda < 0$ and $\lambda = -k^2$. Then the solution to 2.49 is

$$X = Ae^{kx} + Be^{-kx}$$

Using the boundary conditions yield $X(0) = A+B = 0$ and $X(1) = Ae^{k}+Be^{-k} =$

0. Solving the equations $A = B = u = 0$ which is a trivial solution, thus discarded.

- Let $\lambda = 0$, the solution to 2.49 is

$$X(x) = Ax + B$$

The boundary conditions imply $X(0) = B = 0$ and $X(1) = A = 0$. Thus this case is discarded too.

- Finally, let $\lambda > 0$,the solution to 2.49 is

$$X(x) = A\cos(\sqrt{\lambda}x) + B\sin(\sqrt{\lambda}x)$$

The boundary conditions leads to $X(0) = A = 0$ and $X(1) = B\sin(\sqrt{\lambda}) = 0$. Since we do not want a trivial solution where $B = 0$, the equation reduces to

$$\sin(\sqrt{\lambda}) = 0 \tag{2.52}$$

Thus $\sqrt{\lambda} = n\pi$ for $n = 1, 2, 3, ...$ Solution to 2.49 becomes,

$$X_n = b_n \sin(n\pi x), \quad n = 1, 2, 3, ... \tag{2.53}$$

As we determined $\lambda = n^2\pi^2$ for $n = 1, 2, 3, ...$ Solving 2.50 for $T(t)$ gives the solution

$$T^{'}(t) = -n^2\pi^2 T(t) T_n = c_n \exp(-n^2\pi^2 t) \tag{2.54}$$

$$\tag{2.55}$$

where c_n's are integration constants.

Putting the solution of $T(t)$ and $X(x)$ together,

$$u(x,t) = \sum_{n=1}^{\infty} B_n \exp(-n^2\pi^2 t)\sin(n\pi x) \tag{2.56}$$

where we have set $B_n = c_n b_n$. The initial condition gives

$$u(x,0) = \sin(\pi x) = \sum_{n=1}^{\infty} B_n \sin(n\pi x) \tag{2.57}$$

which is a Fourier sine series. Solving for the B_n's, we use the orthogonality property for the eigenfunctions $\sin(n\pi x)$.

$$\int_0^1 \sin(m\pi x)\sin(\pi n x)\mathrm{d}x = \begin{cases} 0, & \text{if } m \neq n \\ 1/2, & \text{m = n} \end{cases} = 0.5\delta_{mn}$$

where δ_{mn} is the kronecker delta,

$$\delta_{mn} = \begin{cases} 0, & \text{if } m \neq n \\ 1, & \text{m = n} \end{cases}$$

Solving 2.57 for B_n, multiplying both sides with $\sin(m\pi x)$ and integrate from 0 to 1 and from the definition of kronecker delta yields

$$B_n = 2\int_0^1 \sin(\pi x)\sin(\pi n x)\mathrm{d}x = \frac{2\sin(\pi n)}{\pi - \pi n^2} \tag{2.58}$$

Combining the solutions

$$u(x,t) = \sum_{n=1}^{\infty} \frac{2\sin(\pi n)}{\pi - \pi n^2} \exp(-n^2\pi^2 t)\sin(n\pi x) = \exp(-\pi^2 t)\sin(\pi x) \tag{2.59}$$

□

2.2.2 Two Dimensional Heat Equation

The natural extension of our study of the one-dimensional partial differential equation problem is to investigate more than one space-like dimensions. When more than one space dimensions are involved, we have to deal with equations such as two-dimensional heat equation or multi-asset Black-Scholes equation. We will consider the following two-dimensional heat equation and conditions for research purposes.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2.60}$$

$$u(x,y,0) = 1, \quad 0 \le x \le x_{\max}, \quad 0 \le y \le y_{\max} \tag{2.61}$$

$$u(x,0,t) = u(x,y_{max},t) = 0, \quad 0 \le t \le T \tag{2.62}$$

$$u(0,y,t) = u(x_{max},y,t) = 0, \quad 0 \le t \le T \tag{2.63}$$

Throughout this work, the case where $T = 0.06$, $x_{max} = 1.0$ and $y_{max} = 1.0$ will be tested.

Theorem 2.2.2. *Similarly to one-dimensional heat equation, applying separation of variables method to the equation yields analytical solution of two-dimensional heat equation.*

Proof. Firstly, variables are separated to divide the equation into simpler problems.

$$\begin{aligned} u(x,t) &= X(x)Y(y)T(t) \\ X^{''}(x) - BX(x) &= 0 \\ Y^{''}(y) - CY(y) &= 0 \\ T^{'}(t) - (B+C)T(t) &= 0 \\ X(0) = X(1) = 0, \quad Y(0) &= Y(1) = 0 \end{aligned}$$

In 2.2.1, we have already seen that the solutions to X(x) and Y(y) are

$$X_m(x) = b_n \sin(m\pi x) \tag{2.64}$$

$$Y_n(x) = a_m \sin(n\pi y) \tag{2.65}$$

Using these values to solve for $T(t)$ gives

$$T_{mn}(t) = c_{mn} \exp(-\pi^2(m^2+n^2)t) \tag{2.66}$$

Substituting the solutions yields

$$u_{mn}(x,y,t) = X_m(x)Y_n(y)T_{mn}(t) \tag{2.67}$$

$$u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} A_{mn}\sin(m\pi x)\sin(n\pi y)\exp(-\pi^2(m^2+n^2)t) \tag{2.68}$$

where $A_{mn} = b_n a_m c_{mn}$. The initial condition gives

$$u(x,y,0) = 1 = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} A_{mn}\sin(m\pi x)\sin(n\pi y) \tag{2.69}$$

which is a double Fourier sine series. Thus, the coefficient A_{mn} is chosen such that

$$A_{mn} = 4\int_0^1\int_0^1 \sin(\pi m x)\sin(\pi n y)\mathrm{d}x\mathrm{d}y = \frac{4(\cos(\pi n)-1)(\cos(\pi m)-1)}{\pi^2 mn} \tag{2.70}$$

Combining the solutions

$$u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} \frac{4(\cos(\pi n)-1)(\cos(\pi m)-1)}{\pi^2 mn}\sin(m\pi x)\sin(n\pi y)e^{-\pi^2(m^2+n^2)t} \tag{2.71}$$

$\square$

2.3 Finite Difference Methods

2.3.1 Discretization

Essentially, solving a PDE is the problem of finding a function which depends on values at infinitely many points. Naturally, the first step of finite difference methods is to make the problem discrete that we can solve [43]. As a result, we need to discretize the space dimensions and time dimension. The discretization procedure begins by replacing the domain $[0, x_{max}]$ x $[0, T]$ by a set of mesh points. In order to get a n x m equally spaced mesh points the step sizes are calculated as $\Delta t = \frac{T}{m}$, $\Delta x = \frac{x_{max}}{n}$.

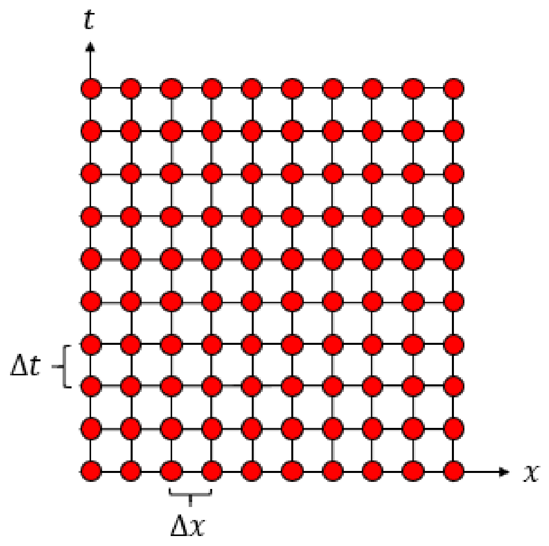
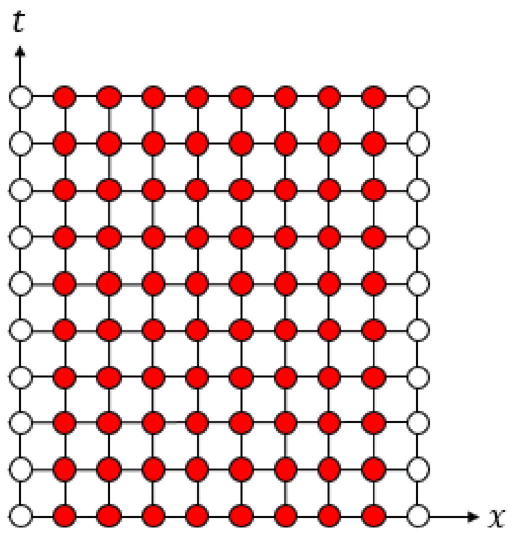


Figure 2.1: Grid where space and time dimensions are discretized by 10 steps.

Following step of discretization process is applying the values given by inital and boundary conditions to the set of points on the grid.



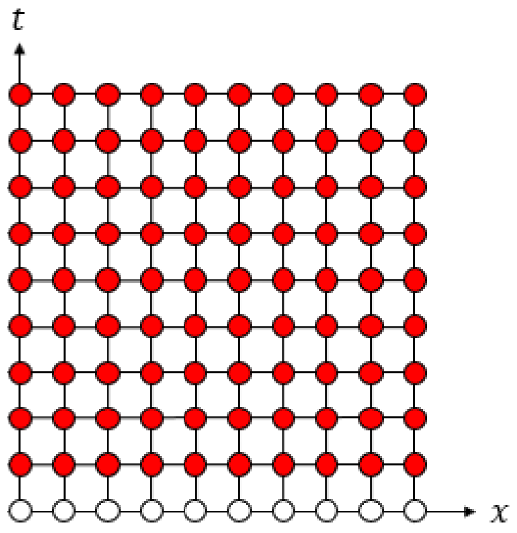


Figure 2.2: Boundary conditions. Figure 2.3: Initial condition.

We need finite difference approximations for the partial derivatives to replace the PDE using the set of points on our grid. Notationally, we will define u_i^n to be a function defined at the point $(i\Delta x, n\Delta t)$.

- Forward difference: $\frac{\partial u}{\partial t} = \frac{u_i^{n+1} - u_i^n}{\Delta t} + \mathcal{O}(\Delta t)$

- Central difference: $\frac{\partial u}{\partial x} = \frac{u^n_{i+1} - u^n_{i-1}}{\Delta t} + \mathcal{O}(\Delta x)$
- Backwards difference: $\frac{\partial u}{\partial x} = \frac{u^n_i - u^n_{i-1}}{\Delta t} + \mathcal{O}(\Delta x)$
- Second order central difference: $\frac{\partial^2 u}{\partial x^2} = \frac{u^n_{i+1} - 2u^n_i + u^n_{i-1}}{(\Delta x)^2} + \mathcal{O}(\Delta x^2)$

As we have a grid of points and approximation of the derivatives, it is possible to obtain a unique solution using numerical methods. The implementation of finite difference method class used in this work can be seen under Appendix B.

2.3.2 Explicit Method

Explicit method generalises the parabolic partial differential equation by applying the forward difference to the time derivative and the centred second difference (FTCS scheme).

$$u_t = a(x,t)u_{xx} + b(x,t)u_x + c(x,t)u \tag{2.72}$$

We will be applying the finite differences to the equation 2.72 for the purposes of simplicity since heat equation and Black-Scholes equation can be generalized in the form for certain choices of coefficients. Applying the forward time and centred space differences where $r = \frac{\Delta t}{\Delta x^2}$.

$$\begin{aligned}
\frac{u^{n+1}_i - u^n_i}{\Delta t} &= a(x,t)\frac{u^n_{i+1} - 2u^n_i + u^n_{i-1}}{(\Delta x)^2} + b(x,t)\frac{u^n_{i+1} - u^n_{i-1}}{\Delta x} + c(x,t)u^n_i \\
u^{n+1}_i &= u^n_{i+1}\left(\frac{-rb(x,t)}{\Delta x} - ra(x,t)\right) \\
&+ u^n_i(1 + 2ra(x,t) - c(x,t)\Delta t) \\
&+ u^n_{i-1}\left(-ra(x,t) + \frac{rb(x,t)}{\Delta x}\right)
\end{aligned}$$

We will replace the coefficients of $u^n_{i+1}, u^n_i, u^n_{i-1}$ terms with γ, α, β respectively. The formula reduces to

$$u^{n+1}_j = \gamma u^n_{j+1} + \beta u^n_j + \alpha u^n_{j-1}. \tag{2.73}$$

The formula expresses one unknown nodal value directly in terms of known nodal values [11]. It can be expanded as

$$u_1^{n+1} = \gamma u_2^n + \beta u_1^n + \alpha u_0^n$$
$$u_2^{n+1} = \gamma u_3^n + \beta u_2^n + \alpha u_1^n$$
$$.$$
$$.$$
$$.$$
$$u_{j-1}^{n+1} = \gamma u_j^n + \beta u_{j-1}^n + \alpha u_{j-2}^n \quad (2.74)$$

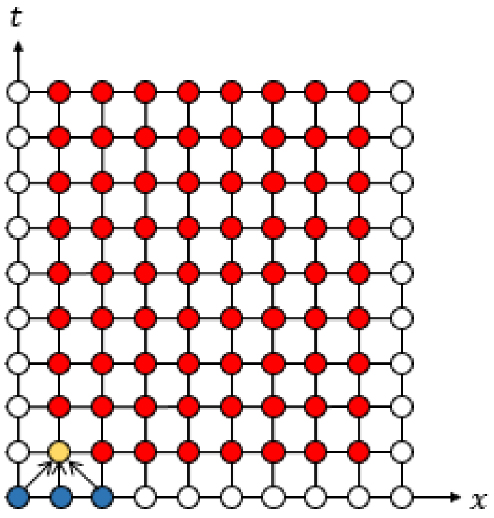


Figure 2.4: Computational stencil of heat equation.

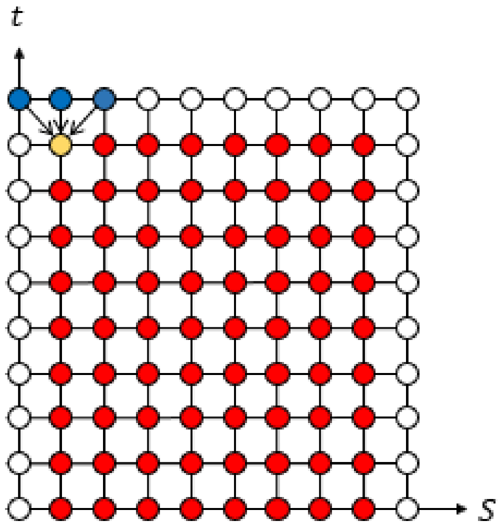


Figure 2.5: Computational stencil of Black-Scholes equation.

Using the boundary conditions and initial condition, the expanded formula can be condensed in the following matrix form.

$$\begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ . \\ . \\ . \\ u_{j-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \alpha u_0^n \\ 0 \\ 0 \\ . \\ . \\ . \\ \gamma u_j^n \end{bmatrix} + \begin{bmatrix} \beta & \gamma & 0 & . & . & 0 \\ \alpha & \beta & \gamma & 0 & ... & . \\ 0 & \alpha & \beta & \gamma & 0 & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & \gamma \\ 0 & 0 & 0 & 0 & \alpha & \beta \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ . \\ . \\ . \\ u_{j-1}^n \end{bmatrix}$$

Deriving the coefficients γ, α, β in the case of heat equation yields

$$\alpha = r \quad \beta = 1 - 2r \quad \gamma = r \tag{2.75}$$

In the case of Black-Scholes formula, since the share price S_j increases linearly with Δx we can replace it as $S_j = j\Delta x$.

$$\alpha = \frac{\sigma^2 j^2 \Delta t}{2} - \frac{rj\Delta t}{2} \quad \beta = 1 - \sigma^2 j^2 \Delta t - r\Delta t \quad \gamma = \frac{\sigma^2 j^2 \Delta t}{2} + \frac{rj\Delta t}{2} \tag{2.76}$$

Lastly, solving heat equation and Black-Scholes differs in time stepping. Black-Scholes formula is solved backwards in time. On the other hand, heat equation is solved forwards in time.

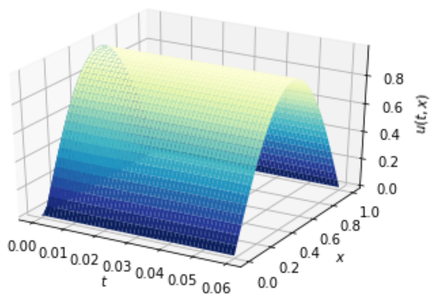


Figure 2.6: Output grid of heat equation using explicit scheme.

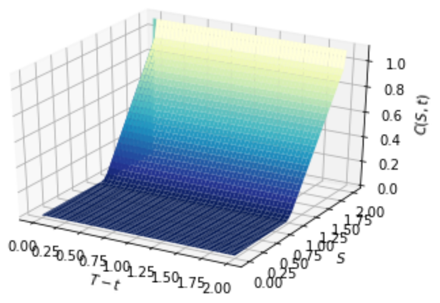


Figure 2.7: Output grid of Black-Scholes equation using explicit scheme.

2.3.3 Crank - Nicolson Method

The explicit method is computationally cheap. However, this brings a serious drawback, for explicit method to attain reasonable accuracy the step size must be kept small [39]. Thankfully, the Crank-Nicolson finite difference scheme was introduced by John Crank and Phyllis Nicolson [7]. In the financial engineering field Crank-Nicolson method is one of the most popular finite difference schemes for calculating numerical solution of the Black-Scholes equation and its variations [40]. If we apply backwards time difference instead of forward time difference that Explicit method used and a central space approximation in space again, we get the BTCS scheme. Applying the BTCS to the base equation 2.72 yields

$$\frac{u_i^{n+1}-u_i^n}{\Delta t} = a(x,t)\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{(\Delta x)^2} + b(x,t)\frac{u_i^{n+1}-u_i^{n+1}}{\Delta t} + c(x,t)u_i^{n+1} \tag{2.77}$$

Crank - Nicolson method takes a weighted average of the FTCS and BTCS schemes. Therefore, the approximations become

$$u(x,t) \approx \frac{1}{2}(u_i^{n+1}+u_i^n) \tag{2.78}$$

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1}-u_i^n}{\Delta t} \tag{2.79}$$

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1}^n - u_{i-1}^n + u_{i+1}^{n+1} - u_{i-1}^{n+1}}{4\Delta x} \tag{2.80}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n + u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{2(\Delta x)^2} \tag{2.81}$$

Applying the new finite differences to the base partial differential equation equation 2.72 yields

$$(-A-B)u_{i+1}^{n+1}+(1+2A-C)u_i^{n+1}+(-A+B)u_{i-1}^{n+1} = (A+B)u_{i+1}^n+(1-2A+C)u_i^n+(A-B)u_{i-1}^n \tag{2.82}$$

where $A = a(x,t)\frac{\Delta t}{\Delta x^2}$, $B = b(x,t)\frac{\Delta t}{4\Delta x}$, $C = c(x,t)\frac{\Delta t}{2}$. Note that in contrast to the FTCS scheme, we now have three unknowns in this equation, the three values of u at the higher time level. We respectively denote the coefficients in the right hand side as γ, β, α and coefficients in the left hand side as λ, θ, ω for simplicity.

$$\lambda u_{i+1}^{n+1} + \theta u_i^{n+1} + \omega u_{i-1}^{n+1} = \gamma u_{i+1}^n + \beta u_i^n + \alpha u_{i-1}^n \tag{2.83}$$

The left hand side groups the unknowns and the right hand side groups knowns. The system of equations can be reduced to a matrix system.

$$\begin{bmatrix} \theta & \lambda & 0 & . & . & 0 \\ \omega & \theta & \lambda & 0 & \dots & . \\ 0 & \omega & \theta & \lambda & 0 & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & \lambda \\ 0 & 0 & 0 & 0 & \omega & \theta \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ . \\ . \\ . \\ u_{j-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \alpha u_0^n \\ 0 \\ 0 \\ . \\ . \\ . \\ \gamma u_j^n \end{bmatrix} + \begin{bmatrix} \beta & \gamma & 0 & . & . & 0 \\ \alpha & \beta & \gamma & 0 & \dots & . \\ 0 & \alpha & \beta & \gamma & 0 & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & \gamma \\ 0 & 0 & 0 & 0 & \alpha & \beta \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ . \\ . \\ . \\ u_{j-1}^n \end{bmatrix}$$

The problem reduces to a tridiagonal matrix system. This system of equations can be solved by various algorithms such as Gaussian elimination or Thomas algorithm.

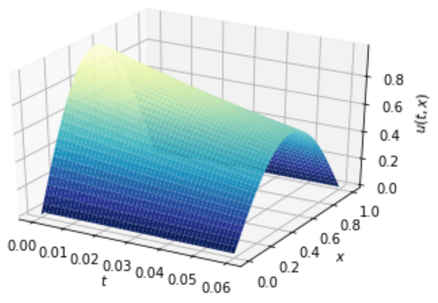


Figure 2.8: Output grid of heat equation using Crank - Nicolson scheme.

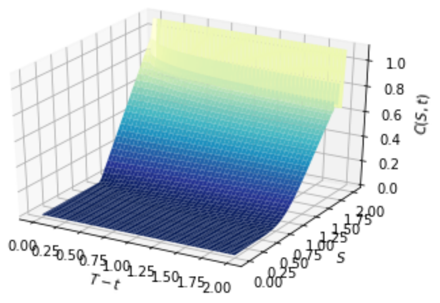


Figure 2.9: Output grid of Black-Scholes equation using Crank - Nicolson scheme.

2.3.4 Alternating Direction Implicit Method

Curse of dimensionality effects finite difference schemes as they tend to become more difficult to set up, understand and implement as the dimensionality of the space increases [9]. The alternating direction implicit (ADI) method is one of the most common techniques to numerically solve two-dimensional parabolic PDEs. The scheme was first proposed by Peaceman and Rachford in 1955 for oil reservoir modelling [35]. The method consists of splitting the time dimension and solving the two-dimensional problem as two consecutive one-dimensional problems. At each time step, the spatial dimensions are solved implicitly in one direction and explicitly in the other dimension. Using the alternating direction implicit scheme provides us with the advantages of implicit method. Computationally requires only solving tridiagonal systems. It is possible to use ADI in more than three dimensions which produce the same number of consecutive one-dimensional problems [8]. To develop a more compact notation, we introduce the finite difference operator notation δ^2.

$$\delta x^2 u_{i,j}^n = \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} \tag{2.84}$$

Explicit method in two space dimensions can be abbreviated as

$$\frac{u_{i,j}^{n+1} + u_{i,j}^n}{\Delta t} = \delta x^2 u_{i,j}^n + \delta y^2 u_{i,j}^n \tag{2.85}$$

and implicit method in two space dimensions can be written as

$$\frac{u_{i,j}^{n+1} + u_{i,j}^n}{\Delta t} = \delta x^2 u_{i,j}^{n+1} + \delta y^2 u_{i,j}^{n+1}. \tag{2.86}$$

Dividing each time step in half we introduce a temporary intermediate unknown $u_{i,j}^{n+1/2}$. Firstly, the two dimensional heat equation is approximating implicitly x and explicitly over y. The total work involved in one time step amounts to solving $N_{steps}-1$ tridiagonal systems [31].

$$\frac{u_{i,j}^{n+1/2} + u_{i,j}^n}{0.5\Delta t} = \frac{\delta x^2 u_{i,j}^{n+1/2}}{\Delta x^2} + \frac{\delta y^2 u_{i,j}^n}{\Delta y^2} \tag{2.87}$$

Rearranging the set of equations yields a tridiagonal system which is solved for the temporary intermediate unknown $u_{i,j}^{n+1/2}$.

$$-r_x u_{i+1,j}^{n+1/2} + (1+2r_x)u_{i,j}^{n+1/2} - r_x u_{i,j}^{n+1/2} = r_y u_{i,j+1}^n + (1+2r_y)u_{i,j}^n + r_y u_{i,j-1}^n \tag{2.88}$$

Next step of the grid $u_{i,j}^{n+1}$ is calculated by approximating explicitly x and implicitly over y.

$$\frac{u_{i,j}^{n+1} + u_{i,j}^{n+1/2}}{0.5\Delta t} = \frac{\delta x^2 u_{i,j}^{n+1/2}}{\Delta x^2} + \frac{\delta y^2 u_{i,j}^{n+1}}{\Delta y^2} \tag{2.89}$$

Rearranging the set of equations yields a tridiagonal system which can be solved using Gaussian elimination, cyclic reduction or Thomas algorithm.

$$-r_y u_{i,j+1}^{n+1} + (1+2r_y)u_{i,j}^{n+1} - r_y u_{i,j-1}^{n+1} = r_x * u_{i+1,j}^{n+1/2} + (1+2r_x)u_{i,j}^{n+1/2} + r_x u_{i,j}^{n+1/2} \tag{2.90}$$

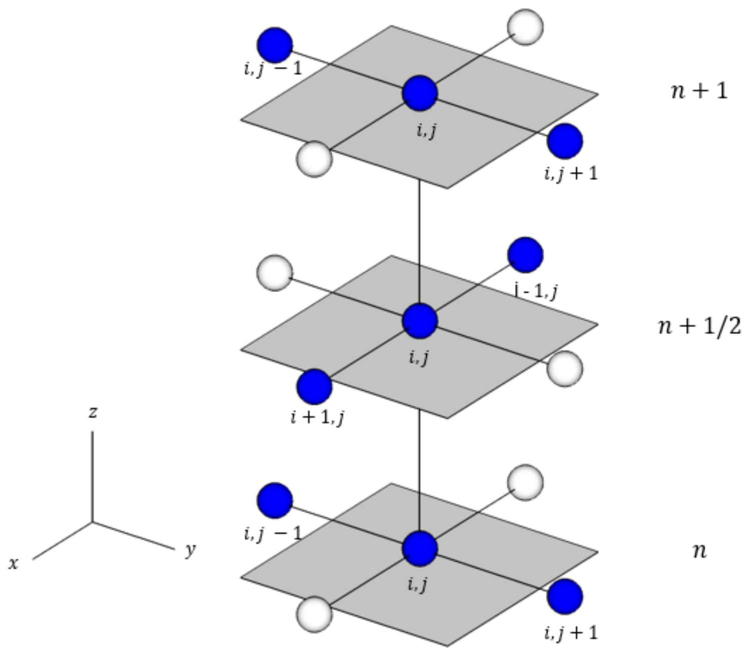


Figure 2.10: Computational stencil of alternating direction implicit method.

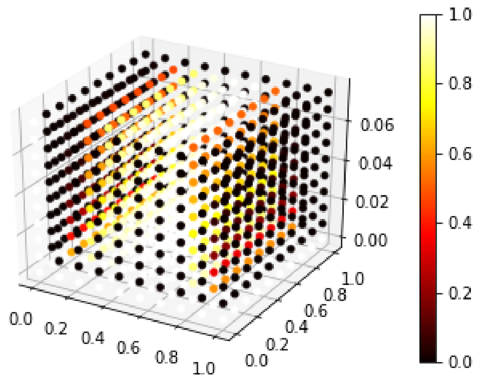


Figure 2.11: Output grid of two dimensional heat equation using ADI.

Chapter 3

Optimizing Solvers

Attempting to progress in solving complex problems using numerical methods is impossible without applying optimizations. Nowadays the difficulty is not solving a given problem but rather solving it in a given computing environment while optimally exploiting the resources. Thus, it is necessary to investigate methods that allow for efficient implementations. This section aims to introduce practical optimization techniques that can be easily implemented on a regular trading floor. Main optimization techniques that will be tested are parallelizing tridiagonal solvers, Visual Studio optimization switches, compilers and solution platforms.

3.1 Solution Platforms

The CPU accesses data from RAM using the register that stores memory addresses. 32-bit and 64-bit refer to the amount of data the system can access. so a 32-bit system can address a maximum of 4 GB ($4,294,967,296$ bytes) of RAM where a 64-bit system can access $18,446,744,073,709,551,616$ bytes of memory. Since 32 bit does not have access to more than 4 GB, if the system has more than 4 GB of RAM, it will be inaccessible by the CPU, thus a 64-bit system will be needed. The memory increase of 64 bit systems means it is capable of very fast processing of numerical operations. One disadvantage of the 64 bit systems is more requirement of memory because addresses are 64 bits (8 bytes) wide instead of 32 bits (4 bytes) wide. Due to the increased sizes, 64-bit programs will occupy more memory than a 32-bit version. Visual Studio offers the x86 and x64 solution platforms which corresponds to 32-bit and 64-bit respectively. The solution platforms will be tested against to determine the optimal solution platform.

3.2 Compilers

The software we write is translated into low-level abstractions by a compiler. The quality of translation plays a crucial role in how the software performs. Commonly, compilers are comprised of three stages, front end, optimization and the back end. The first step is to understand the source code to translate it into intermediate representation using data structures and formal language theory. The intermediate representation generated in the front end is later used by the back end. In the middle, the optimizer is focused on efficiency. It transforms the intermediate representation by deriving knowledge about runtime behaviour and improve the behaviour. Finally, the back end maps the functionality to the instruction set of the processor [45].

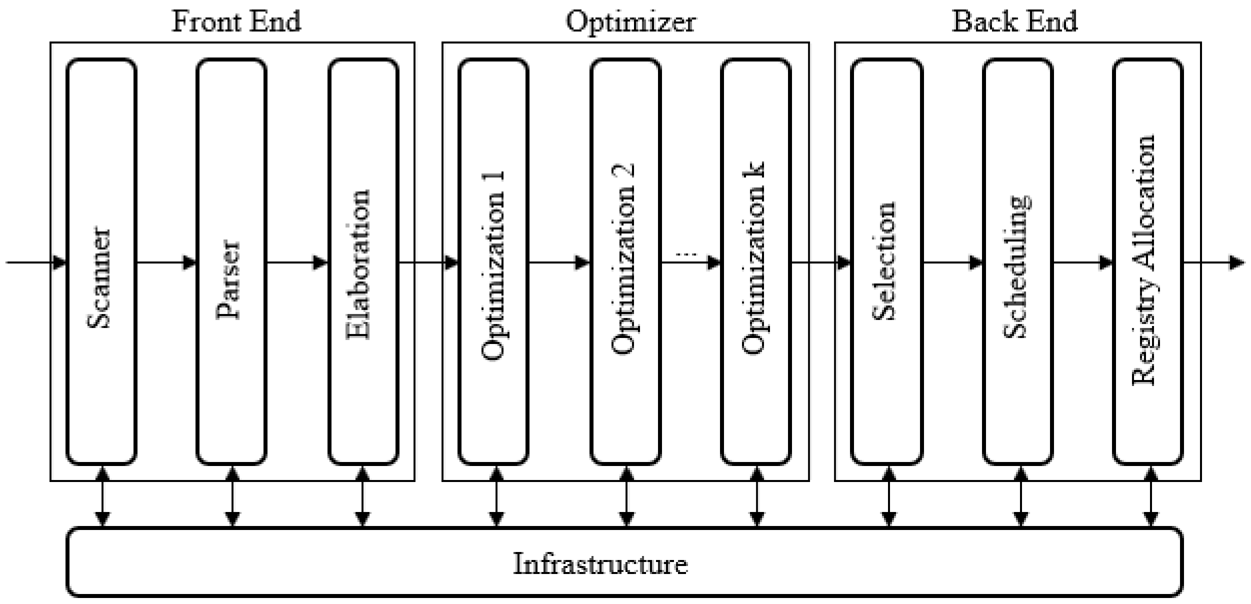


Figure 3.1: Basic structure of a compiler.

During the back end stage, the compiler approximates the allocation and scheduling. The speed and size of the code is a direct result of the ability to approximate correctly. This produces complex interactions that can lead to problematic results. Therefore, regardless of the implementation of the program, the performance can be different under varying compilers and they are an important factor for time-constrained tasks such as option pricing. Well designed and implemented compilers savings accumulate over time. It should be able to produce well-optimized code and let us focus on the process of writing programs rather than struggling with the inadequacies of the compiler. In this project, Visual C++ and Intel C++ compiler will be tested against each other.

3.3 Visual Studio Optimization Switches

Visual Studio Optimization Switches, also known as /O options controls various optimizations to be chosen according to the needs of the project. There are various switches for different goals such as minimizing the size of the code (/O1) but since the scope of this project is limited with speed optimizations. Speed optimization flags are /O2 and /Ox. /O2 is a combination of /Og, /Oi, /Ot, /Oy, /Ob2, /GF and /Gy flags. /Ox is a subset of /O2 without the /GF and /Gy flags. These additional options applied by /O2 can cause pointers to strings or to functions to share a target address, which can affect debugging and strict language conformance [28].

- /Og: Enables local and global optimizations (subexpression elimination), automatic-register allocation, and loop optimization [29].
- /Oi: Generates intrinsic functions for appropriate function calls. Compiler may not replace the function call with an intrinsic if it will result in better performance [26].
- /Ot: Favors optimizations for speed over optimizations for size by instructing the compiler to reduce many C and C++ constructs to functionally similar sequences of machine code. If /Ot is used, /Og must be specified to optimize the code [27].
- /Oy: Suppresses the creation of frame (base) pointers on the call stack for quicker function calls. Frees one register for general usage [30].
- /Ob2: Controls inline expansion of functions. Under /O2 and /Ox, allows the compiler to expand any function including the ones that are not explicitly marked for no inlining. Function-calling-overheads are saved thus inline functions run faster than the normal functions with a memory penalty [25].
- /GF: Enables the compiler to create a single copy of identical strings in the program image and in memory during execution. This is an optimization called string pooling that can create smaller programs. Under this flag, strings are pooled as read-only, trying to modify strings throws an error [23].

- /Gy: Enables function-level linking. Allows the compiler to package individual functions in the form of packaged functions (COMDATs) or order individual functions in a DLL or .exe file [24].

/O2 and /Ox flags are tested for maximum speed against the /Od flag which disables all the optimizations.

3.4 Tridiagonal Solvers

Tridiagonal solvers are the most demanding part of the solvers. Hence, development and improvement of such solvers is of great interest [41] [6] [32] [34] concerned with this problem. Large tridiagonal systems appear in many numerical analysis applications. In our work, they arise in the Crank-Nicolson and Alternating Direction Implicit schemes. Solving tridiagonal systems is the most computationally intensive part of the schemes. Therefore, choosing efficient tridiagonal solvers is crucial for the speed of the solver. In this experiment, implementations of the Thomas algorithm, Gaussian elimination and cyclic reduction will be tested.

3.4.1 Thomas Algorithm

Thomas Algorithm is the most commonly used method for solving a tridiagonal system of equations. The method is used to solve a tridiagonal matrix system invented by Llewellyn Thomas [44]. The algorithm is equivalent to Gaussian elimination without pivoting. The system equations can be written as

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1} \\ 0 & 0 & 0 & 0 & a_k & b_k \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_k \end{bmatrix}$$

The method begins by calculating coefficients c_i^* and d_i^* replacing a_i, b_i and c_i [10].

$$c_i^* = \begin{cases} \frac{c_1}{b_1} & ; i = 1 \\ \frac{c_i}{b_i - c_{i-1}^* a_i} & ; i = 2, 3, ..., k-1 \end{cases}$$

$$d_i^* = \begin{cases} \frac{d_1}{b_1} & ; i = 1 \\ \frac{d_i - d_{i-1}^* a_i}{b_i - c_{i-1}^* a_i} & ; i = 2, 3, ..., k-1 \end{cases}$$

The equations can be rewritten as

$$\begin{bmatrix} 1 & c_1^* & 0 & 0 & ... & 0 \\ 0 & 1 & c_2^* & 0 & ... & 0 \\ 0 & 0 & 1 & c_3^* & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1}^* \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1^* \\ d_2^* \\ d_3^* \\ . \\ . \\ . \\ d_k^* \end{bmatrix}$$

The last step is to work in reverse with the following equations.

$$f_k = d_k^*, \qquad f_i = d_k^* - c_i^* x_{i+1}, \qquad i = k-1, k-2, ..., 2, 1$$

Notice that when solving Thomas Algorithm, it cannot take advantage of parallelism as each step depends on the other element.

3.4.2 Intel Math Kernel Library

Intel Math Kernel Library implements routines for solving systems of linear equations from the standard LAPACK library which is a software package provided by the University of Tennessee. Variety of matrix types are supported by the routines. Specifically, gtsv function is utilized from the package. Using Gaussian elimination with partial pivoting, gstv computes the solution to the system of linear equations with a tridiagonal coefficient matrix [16]. Gaussian elimination with partial pivoting starts by determining the pivot by finding the largest absolute value at the left column. If necessary, row interchange is performed to ensure that the largest value is at the first-row [46].

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1} \\ 0 & 0 & 0 & 0 & a_k & b_k \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_k \end{bmatrix}$$

Assume that the first row is the pivot in the above system of equations. The pivot row is substituted from the second row to eliminate a_2. The process of selecting pivot and eliminating an element is repeated until the system of equations is in upper triangular form

$$\begin{bmatrix} b_1^* & c_1^* & 0 & 0 & ... & 0 \\ 0 & b_2^* & c_2^* & 0 & ... & 0 \\ 0 & 0 & b_3^* & c_3^* & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1}^* \\ 0 & 0 & 0 & 0 & 0 & b_k^* \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1^* \\ d_2^* \\ d_3^* \\ . \\ . \\ . \\ d_k^* \end{bmatrix}$$

Once the system is in this form, it is easily solved by back substitution.

3.4.3 Cyclic Reduction

Cyclic reduction was proposed by R. W. Hockney in the 1960s for solving the resulting linear systems from the discretization of the Poisson equation [14]. Cyclic reduction consists of two stages, forward reduction and back substitution. Given an n sized system, we will model the matrix as three vectors, lower diagonal `LDiag` of size $n-1$, diagonal `Diag` of size n and upper diagonal `UDiag` of size $n-1$.

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1} \\ 0 & 0 & 0 & 0 & a_k & b_k \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_k \end{bmatrix}$$

During the forward reduction stage, at each step s all odd-indexed equations are updated in parallel using the following equations.

$$a_i^s f_{i-2^{s-1}}^s + f_i^s + c^s x_{i+2^{s-1}} = d_i^s \tag{3.1}$$

are generated where $i = 2^s, 2^s + 2^s, ..., n$ and $s = 1, 2, ..., \log_2 n$. In the implementation the i and s are respectively denoted by the variables `iReduce` and `Step`.

```
for (Step = 1; Step <= int(log2(size)); Step++)
{
  for (int iReduce = pow(2, Step) - 1; iReduce < Diag.size();
  iReduce += pow(2, Step))
  {
      offset = pow(2, Step - 1);
      iReduceMinus = iReduce - offset;
      iReducePlus = iReduce + offset;
```

The updated values at each step are

$$k_1 = \frac{a_i^{s-1}}{b_{i-1}^{s-1}}, \qquad k_2 = \frac{c_i^{s-1}}{b_{i+1}^{s-1}} \tag{3.2}$$

$$a_i^s = -a_{i-1}^{s-1} k_1, \qquad c_i^s = -c_{i+1}^{s-1} k_2 \tag{3.3}$$

$$b_i^s = b_i^{s-1} - c_{i-1}^{s-1} k_1 - a_{i+1}^{s-1} k_2 \tag{3.4}$$

$$d_i^s = d_i^{s-1} - d_{i-1}^{s-1} k_1 - d_{i+1}^{s-1} k_2 \tag{3.5}$$

Thus, the number of unknowns are reduced by half at each step. The same procedure is applied recursively until there remains only one equation with one unknown where $i = 2^s, 2^s + 2^s, ..., n$ and $s = 1, 2, ..., \log_2 n$. Since we can't determine $i + 1$ when we are updating the last equation, it is treated differently. The loop in the source code continues to update the values with the given equations where k_1 and k_2 values are stored in the `Temp1` and `Temp2` variables. The `oldResult` vector is used to store the known values and `newResult` is used to store the solution for the unknowns.

```
      //Last equation is treated differently
      if (iReduce == Diag.size() - 1)
      {
          Temp1 = LDiag[iReduce] / Diag[iReduceMinus];
          LDiag[iReduce] = -LDiag[iReduceMinus] * Temp1;
          UDiag[iReduce] = 0;
          Diag[iReduce] = Diag[iReduce] - (UDiag[iReduceMinus] * Temp1);
          oldResult[iReduce] = oldResult[iReduce] - oldResult[iReduceMinus]
              * Temp1;
      }
      else
      {
          Temp1 = LDiag[iReduce] / Diag[iReduceMinus];
          Temp2 = UDiag[iReduce] / Diag[iReducePlus];

          LDiag[iReduce] = -LDiag[iReduceMinus] * Temp1;
          UDiag[iReduce] = -UDiag[iReducePlus] * Temp2;
          Diag[iReduce] = Diag[iReduce] - (LDiag[iReducePlus] * Temp2)
               - (UDiag[iReduceMinus] * Temp1);
          oldResult[iReduce] = oldResult[iReduce]
              - (oldResult[iReduceMinus] * Temp1)
              - (oldResult[iReducePlus] * Temp2);
      }
  }
}
```

After the forward reduction is finished. Backward substitution phase starts where we solve all rest of the unknowns by substituting the already solved value at each time step. As the name implies the steps go backwards as $s = \log_2 n - 1, ..., 0$ and $i = 2^s, 2^s + 2^{s+1}, ..., n$. Similarly to the forward reduction phase, the i and s variables

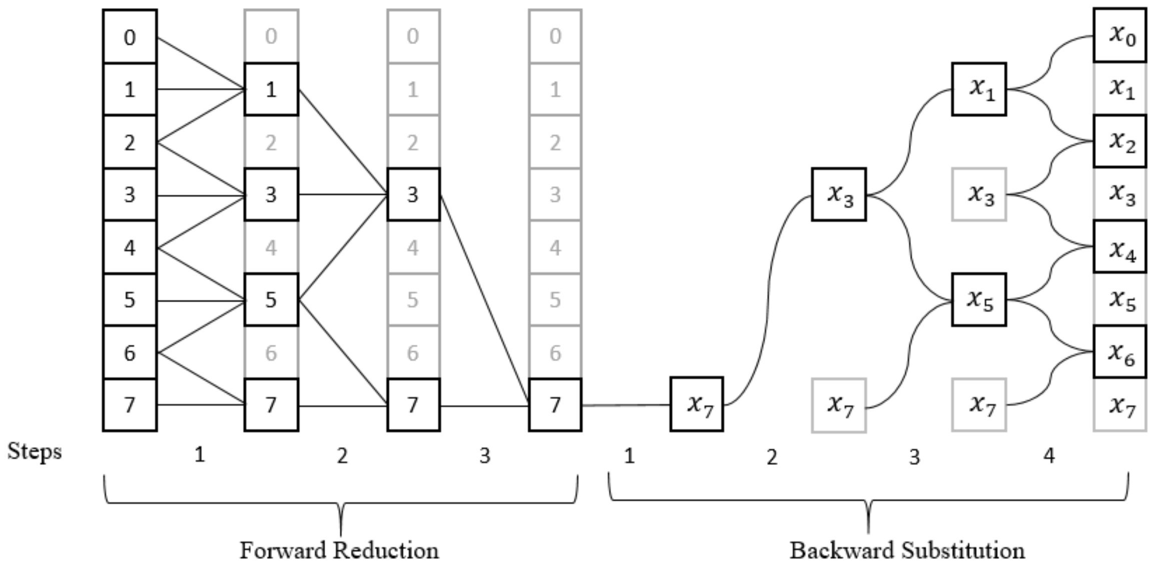


Figure 3.2: Cyclic reduction for an eight equation system.

are stored as `backSub` and `Step`.

```
for (int Step = log2(size + 1) - 2; Step >= 0; Step--)
{
    for (int backSub = pow(2, Step + 1) - 1; backSub < size; backSub += pow(2, Step +
    {
        offset = pow(2, Step);
        backSubMinus = backSub - offset;
        backSubPlus = backSub + offset;
```

The values are determined by the formula at each time step.

$$f_i = \frac{d_i^s - a_i^s f_{i-2^s} - c_i^s f_{i+2^s}}{b_i^s} \tag{3.6}$$

However, the last and the first equations require a special treatment since they do not have a next or previous element to calculate their solution. The rest of the loop continues as

```
        //Special treatment of the first element.
        if (backSubMinus - offset < 0)
        {
            newResult[backSubPlus] = (oldResult[backSubPlus]
                - (UDiag[backSubPlus] * newResult[backSubPlus + offset]))
                / Diag[backSubPlus];
        }
        //Special treatment of the last element.
        else if (backSubPlus + offset >= size)
        {
            newResult[backSubMinus] = (oldResult[backSubMinus]
                 - (newResult[backSubMinus - offset] * LDiag[backSubMinus]))
                 / Diag[backSubMinus];

        }
        else
        {
            newResult[backSubPlus] = (oldResult[backSubPlus]
                - (newResult[backSubPlus - offset] * LDiag[backSubPlus])
                - (UDiag[backSubPlus] * newResult[backSubPlus + offset]))
                 / Diag[backSubPlus];
            newResult[backSubMinus] = (oldResult[backSubMinus]
            - (newResult[backSubMinus - offset] * LDiag[backSubMinus])
            - (UDiag[backSubMinus] * newResult[backSubMinus + offset]))
            / Diag[backSubMinus];
        }
    }
}
```

If serially computed, Thomas algorithm performs $8n$ operations while cyclic reduction performs $17n$ operations. On the other hand,if parallel computing is used with n cores, cyclic reduction requires $2\log_2 n - 1$ steps while the Thomas algorithm requires $2n$ steps [50]. The cyclic reduction algorithm was focused towards fine-grained parallelism which could be achieved using CPU parallelism [37].

3.5 Open Multi-Processing

Traditionally, programs are serially computed on a single processor. On the other hand, parallel computation is used to break our code execution in pieces so that it utilizes parallelism. Multithreading uses the CPUs cores to run calculations concurrently in each core. The concurrent programs are called a thread. If the code is executed on parallel processors, one of the biggest problems is the processors generally require results that have been calculated on other processors. The main issue, in this case, is that processors clocks are not synchronized and execute the code at minimally different speeds.

In order to solve this problem, a group of major computer hardware and software vendors and major parallel computing user facilities joined forces to form The Open Multi-Processing Architecture Review Board (The OpenMP ARB)[42]. Open Multi-Processing (OpenMP) is an implementation of multithreading for C, C++ and Fortran and it was introduced to public in 1997. OpenMP is aiming to standardize high level parallelism that is performant, productive and portable.

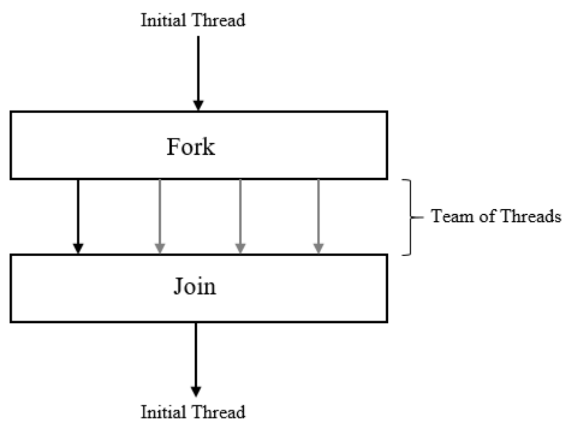


Figure 3.3: The fork-join programming model.

OpenMP approach to multithreading is the fork-join programming model. Firstly, the program start as a single thread of execution called the initial thread. The fork stage begins when the program encounters an OpenMP parallel construct. Parallel execution takes place and multiple threads are created in the parallel region. The initial thread becomes the master and collaborates with the newly created threads to execute the code dynamically. Finally, at the join stage all threads are synchronized, threads are terminated except the original thread [4].

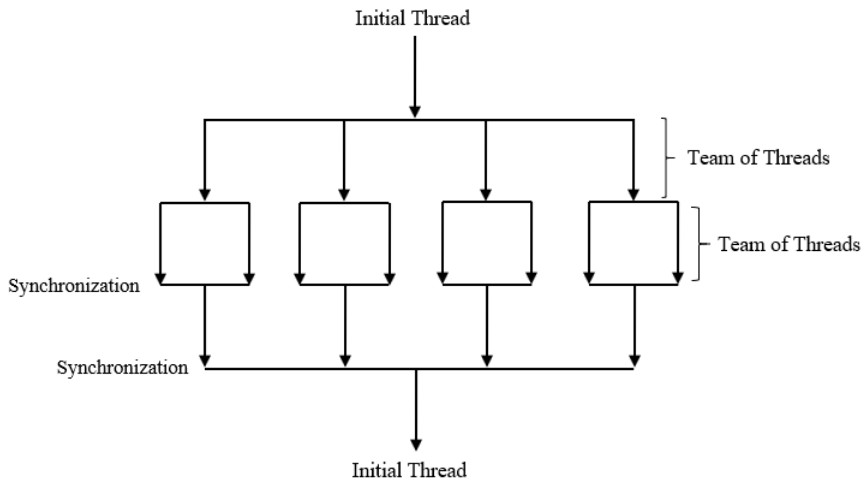


Figure 3.4: 4 threads create 2 threads, nested parallelism.

Algorithms such as cyclic reduction 3.4.3 uses nested loops to calculate the solution. Therefore, nested parallelism is needed to build efficient programs which is achieved by nesting the parallel constructs. Each thread that encounters the next parallel region creates a new parallel region at runtime [47].

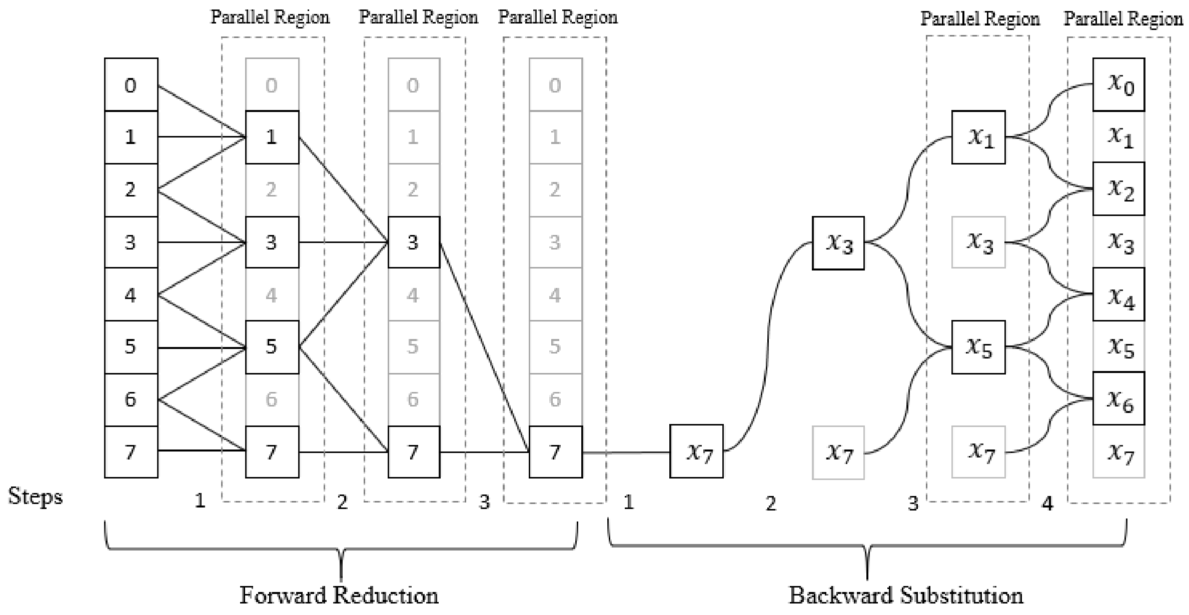


Figure 3.5: Implementation of cyclic reduction using OpenMP.

In order to specify and control the parallelization procedure, OpenMP uses compiler directives, runtime functions, and environment variables. OpenMP enables developers to just give a high-level specification of the parallelism by indicating the regions to be executed in parallel using compiler directives, runtime library routines, and environment variables. The details of the parallelism are up to the compiler which makes OpenMP comparatively easy to use.

3.6 Timing the Code

Timers and clocks might differ on distinct systems. Therefore, measuring execution time intervals accurately is an important task to compare the efficiency and speed of different environments and implementations.

3.6.1 Windows Application Programming Interface

Windows Application Programming Interface (API) is the lowest level of interaction between applications and the Windows operating system. Thus every program is built upon the API. Mostly, the interaction is hidden, the runtime and support libraries manage it in the background [36]. The APIs can be used in the C++ environment. Runtime can be calculated by ”QueryPerformanceCounter” or ”QueryPerformanceFrequency” functions. Respectively, the functions retrieve a high resolution time stamp and the frequency of the performance counter.

3.6.2 Chrono Library

Given the amount of work, using the Windows API for just timing the code is slightly excessive and dependent on the Windows platform. Chrono library was introduced part of the C++11's standard library. The Chrono library is intended to work with date and time. Chrono library is advantageous since it is platform independent and easy to implement. The `high_resolution_clock` provides the smallest possible tick period and with the `now` method, returns a value corresponding to the call's point in time.

```
auto Beginning = chrono::high_resolution_clock::now();
    //Portion of code to be timed
auto End = chrono::high_resolution_clock::now();
```

Once the start and end time of the code is recorded, the duration::count method is used to get the elapsed time. The default setting returns the duration in seconds. Using the `duration\_cast`, the resolution can be converted in hours, minutes, seconds, milliseconds, microseconds and nanoseconds.

```
chrono::duration<double> Elapsed = End - Beginning;
// Prints the time in seconds.
cout << "Elapsed time:" << Elapsed.count() << std::endl;
// Prints the time in nanoseconds.
cout << chrono::duration_cast<chrono::nanoseconds>(Elapsed).count();
```

An average of 1000 trials is taken to make sure the integrity of timings. Because of the fact that the compiler is clever enough to notice same calculations are repeated, during each execution a random number $0 < \epsilon < 10^{-7}$ is added to the step size.

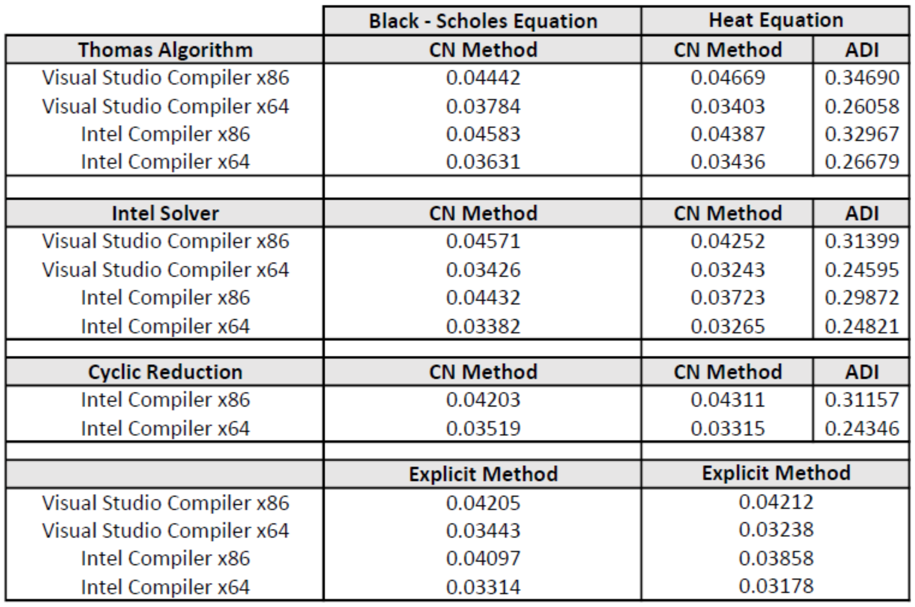
Chapter 4

Results and Discussion

This section documents the performance of attempted optimizations. Experiments are conducted at W307 computer laboratory, Queen Mary University of London. Each computer has Windows 10 Enterprise 64 bit, 16 GB of RAM, Intel Core i7 - 6700 CPU with 4 cores clocked at 3.40 GHz. The source code is written in C++ and compiled with Microsoft Visual Studio Enterprise 2017, Version 15.3.3 in the release mode. External tools utilized in the tests include Intel Compiler, version 18.0.3 and Intel Math Kernel Library. First step was solving the base cases 2.1.4, 2.40, 2.60 by hand and Excel. Following the simple implementations, the solvers are ported to C++ to measure and optimize the performance. Different solution platforms, compilers, optimization flags and tridiagonal solvers are tested against each other.

4.0.1 Base Case

Previously defined analytical solutions 2.1.6, 2.2.1, 2.2.2 are used to calculate errors for the solutions using different grid sizes. Space and time dimensions are discretized by 64 for Explicit and Crank - Nicolson schemes. In order to solve two dimensional heat equation using the ADI scheme two space dimensions and the time dimension are discretized by 32. All optimizations of Visual Studio (/Od flag) are disabled for the base case. Following sections summarizes the performance increases by taking average of runtimes. For detailed timings, please refer to the Appendix C.

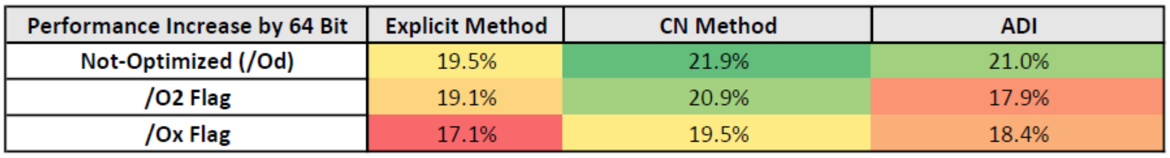


	Black - Scholes Equation	Heat Equation	
Thomas Algorithm	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04442	0.04669	0.34690
Visual Studio Compiler x64	0.03784	0.03403	0.26058
Intel Compiler x86	0.04583	0.04387	0.32967
Intel Compiler x64	0.03631	0.03436	0.26679
Intel Solver	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04571	0.04252	0.31399
Visual Studio Compiler x64	0.03426	0.03243	0.24595
Intel Compiler x86	0.04432	0.03723	0.29872
Intel Compiler x64	0.03382	0.03265	0.24821
Cyclic Reduction	**CN Method**	**CN Method**	**ADI**
Intel Compiler x86	0.04203	0.04311	0.31157
Intel Compiler x64	0.03519	0.03315	0.24346
	Explicit Method	**Explicit Method**	
Visual Studio Compiler x86	0.04205	0.04212	
Visual Studio Compiler x64	0.03443	0.03238	
Intel Compiler x86	0.04097	0.03858	
Intel Compiler x64	0.03314	0.03178	

Figure 4.1: Time taken to solve Black - Scholes equation and heat equation using the Explicit, Crank Nicolson and ADI schemes without any optimization.

4.0.2 Solution Platforms

Switching the solution platform from x86 to x64 resulted in $17 - 22\%$ decrease in the computing time on average. The x86 CPU architecture provides 8 32-bit general registers and 8 SSE registers. Using a x64 architecture increases the number of both registers to 16. Therefore, calculations are more efficient and fast under 64-bit and we will prefer 64-bit for our best case.

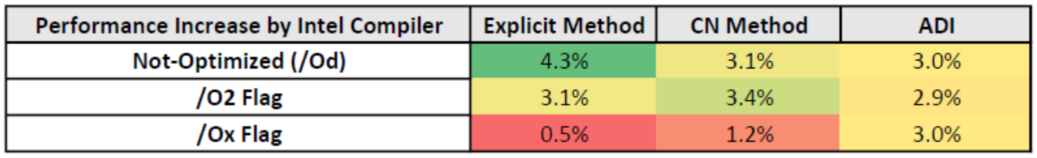


Performance Increase by 64 Bit	Explicit Method	CN Method	ADI
Not-Optimized (/Od)	19.5%	21.9%	21.0%
/O2 Flag	19.1%	20.9%	17.9%
/Ox Flag	17.1%	19.5%	18.4%

Figure 4.2: Average decrease in computation time when switched to x64 solution platform.

4.0.3 Compilers

Switching the Visual Studio compiler to Intel compiler provided $0.5 - 4.3\%$ performance increase.



Performance Increase by Intel Compiler	Explicit Method	CN Method	ADI
Not-Optimized (/Od)	4.3%	3.1%	3.0%
/O2 Flag	3.1%	3.4%	2.9%
/Ox Flag	0.5%	1.2%	3.0%

Figure 4.3: Average performance increase when switched to Intel compiler.

The gtsv function from Intel Math Library was faster under the Intel compiler when compiler optimizations are disabled. However when we enable the /O2 and /Ox optimizations the difference was negligible. The OpenMP support of Visual Studio compiler was not supporting nested for loops, therefore it is only tested using Intel Compiler. All in all, Intel compiler is faster and provides the support for new technologies such as parallelism and should be used when dealing with heavy numerical operations.

4.0.4 Visual Studio Optimizations

The runtime when solving tridiagonal systems greatly benefits from enabling Visual Studio optimizations /Ox and /O2 flags. The Thomas algorithm benefits the most by 10.1% decrease in runtime on average of 24 observations. Similarly, the timings for cyclic reduction improved by 9.5% on average of 12 observations. Given the fact that the functions from Intel Math Library are already heavily optimized, Intel solver improved only 4.6% on average of 24 observations.

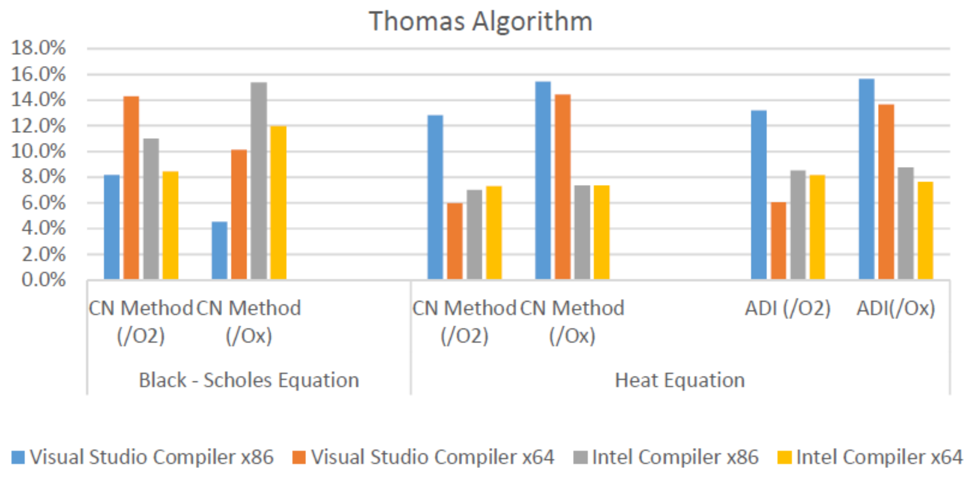


Figure 4.4: Performance improvement of Thomas algorithm under different optimization switches.

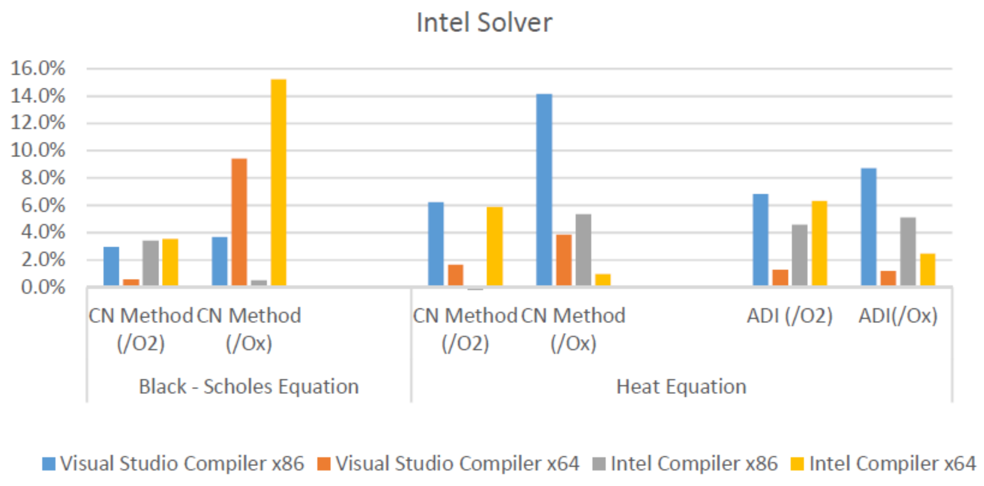


Figure 4.5: Performance improvement of Intel solver under different optimization switches.

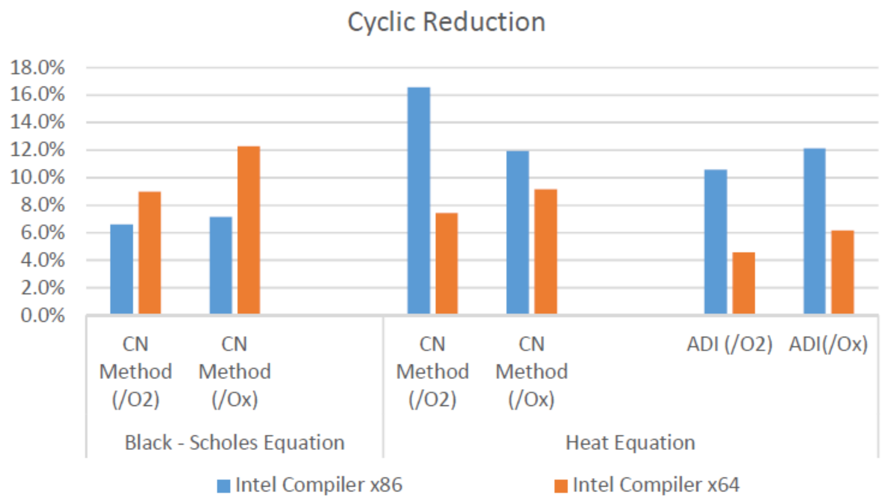
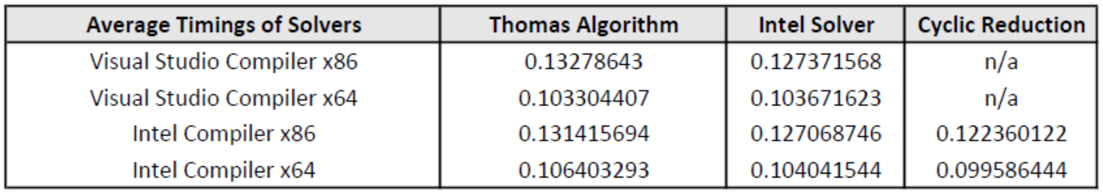


Figure 4.6: Performance improvements of cyclic reduction under different optimization switches.

The /Ox switch recorded the fastest times when solving Black - Scholes equation and heat equation regardless of the tridiagonal solvers. Even the explicit method which doesn't require any matrix operations benefited from switching the optimization switches on. The most performance gain recorded by /Ox flag is 13.8%. Overall explicit methods performance improved by 5.4%. Even though the two optimization switches are really close to each other, overall we can conclude that the /Ox flag performs better in the given task of optimizing schemes. For a detailed view of the performance increases please see the C.5.

4.0.5 Tridiagonal Solvers

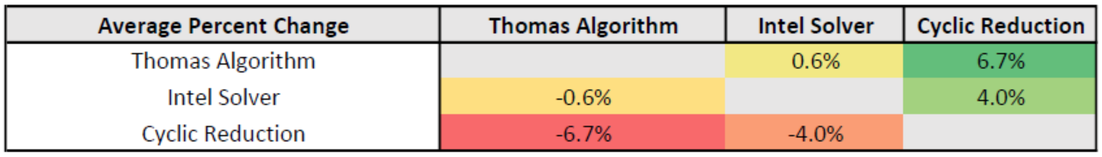
Solving tridiagonal systems is the computationally heaviest component of when calculating solutions with Crank - Nicolson and ADI schemes. Therefore, choosing a fast solver is crucial in our task.



Average Timings of Solvers	Thomas Algorithm	Intel Solver	Cyclic Reduction
Visual Studio Compiler x86	0.13278643	0.127371568	n/a
Visual Studio Compiler x64	0.103304407	0.103671623	n/a
Intel Compiler x86	0.131415694	0.127068746	0.122360122
Intel Compiler x64	0.106403293	0.104041544	0.099586444

Figure 4.7: Average runtime of tridiagonal system solvers.

Compared to our base case, thomas algorithm, cyclic reduction improved the performance 6.7% on average. The gtsv function from the Intel Math Library only improved by 0.6%.



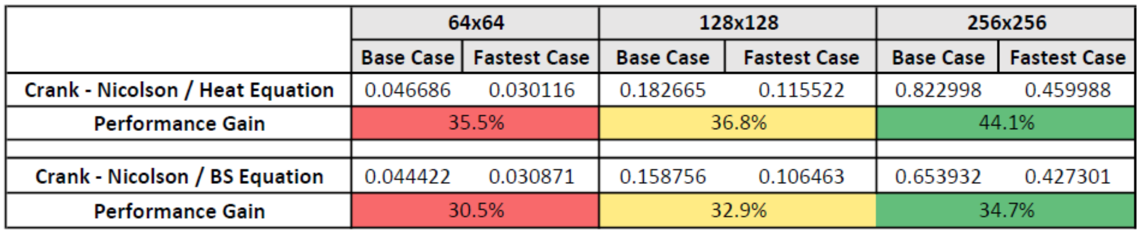
Average Percent Change	Thomas Algorithm	Intel Solver	Cyclic Reduction
Thomas Algorithm		0.6%	6.7%
Intel Solver	-0.6%		4.0%
Cyclic Reduction	-6.7%	-4.0%	

Figure 4.8: Decrease in runtime compared to the base case with Thomas algorithm.

Our base case is using x86 platform, Visual Studio compiler, Thomas algorithm and no optimizations. Under these conditions heat equation was solved in 0.046 seconds and Black - Scholes equation was solved in 0.044 seconds using the Crank - Nicolson method. The fastest time we achieved is 0.030 for heat equation and 0.031 for Black - Scholes equation. Utilizing the Intel compiler, cyclic reduction with OpenMP and /Ox optimization flag resulted in 35.5% and 30.5% performance gain for heat equation and Black - Scholes equation. Under the same base case and fastest case conditions the ADI method for two-dimensional heat equation ran in 0.34 and 0.23 seconds, resulting in 34.1% faster solutions.

4.0.6 Different Sized Discretizations

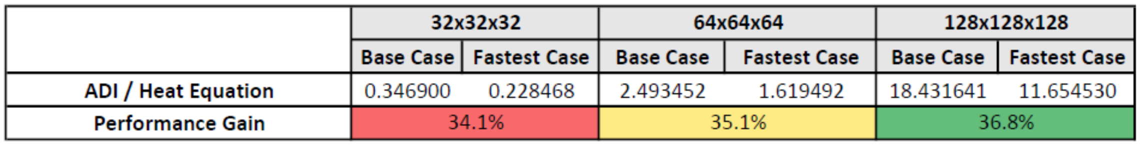
In the numerical experiments above, the grid size is kept constant. Logically, the performance increase should be greater in case of bigger grids. Therefore, the grid size for Crank - Nicolson method was increase to 128 by 128 and 256 by 256. Only the base case and fastest case is tested in this section.



	64x64		128x128		256x256	
	Base Case	Fastest Case	Base Case	Fastest Case	Base Case	Fastest Case
Crank - Nicolson / Heat Equation	0.046686	0.030116	0.182665	0.115522	0.822998	0.459988
Performance Gain	35.5%		36.8%		44.1%	
Crank - Nicolson / BS Equation	0.044422	0.030871	0.158756	0.106463	0.653932	0.427301
Performance Gain	30.5%		32.9%		34.7%	

Figure 4.9: Improvements in runtime between the base case and best case utilizing Crank - Nicolson under different grid sizes.

As expected, the bigger grid size benefits more from the performance gain. The grid size of ADI method is increased to 64 and 128 from 32 for all dimensions.



	32x32x32		64x64x64		128x128x128	
	Base Case	Fastest Case	Base Case	Fastest Case	Base Case	Fastest Case
ADI / Heat Equation	0.346900	0.228468	2.493452	1.619492	18.431641	11.654530
Performance Gain	34.1%		35.1%		36.8%	

Figure 4.10: Improvements in runtime between the base case and best case utilizing ADI under different grid sizes.

Similarly, as the grid size increases the performance gains increase when using ADI method.

Chapter 5

Conclusion

Option pricing is a central component for financial corporations, efficient and accurate pricing of them is critical in day to day operations. For most partial differential equations, there is no analytical formula giving their price, therefore, numerical methods used to price them. In this thesis, we choose the value options and solve the heat equation to test optimizations for the numerical methods, Crank Nicolson method, explicit method and alternating direction implicit scheme. Numerical analysis and computer simulations are undertaken to put theory and observation together to gain insight into the workings of numerical solutions of partial differential equations.

The tests in this work suggest that the Intel compiler with 64 bit produces faster code. The Intel compiler provides detailed documentation and support for libraries and tools to develop high-performance programs such as Intel Math Library, Intel Advisor, Intel VTune Performance Analyzer, etc. The timings suggest that the /Ox optimization switch is generally able to provide the best performance. In implementing finite difference methods, we've verified that solving tridiagonal systems is the bottleneck. Therefore, we studied the effects of various tridiagonal solvers on the speed of the numerical solution. Our results indicate that parallel cyclic reduction is the fastest when using Intel compiler with 64 bit and /Ox switch enabled. Against the base case, the best case decreases the timings for Black-Scholes equation and heat equation by 35.5% and 30.5%. In the two-dimensional heat equation case, the fastest solution recorded a 34.1% performance gain. Lastly, the effects of optimizations when the grid size changes were studied.

5.0.1 Further Work

One of the limitations of this project was the type of options. Changing the type of options such as interest rate derivatives and the Black-Scholes model to another pricing framework such as multi-asset Black-Scholes or HJM model [19] can be implemented. Many optimization techniques have been left for future due to time constraints. The following ideas can provide potential performance improvements

- Numerical methods can be implemented using x86-64 Assembly language and the AVX, AVX2, and AVX-512 instruction set [20].
- General Purpose GPUs can be used to calculate numerical solutions with CUDA or Open Computing Language(OpenCL) but can be challenging because of the requirement of delicate memory management.
- Public cloud-computing services have steadily become cheaper and popular. A cloud function is a serverless execution environment which serves a single function. Building programs on a number of cloud functions is a difficult process. A new study from researchers at Stanford University claims to execute and parallelize existing programs using the cloud functions [12]. Potentially, parallelizing the numerical methods in the cloud can save time and money for financial institutions.

Appendix A

Implementation of the `PDE` class

Throughout the project, parabolic partial differential equations are generalised as

$$u_t = a(x,t)u_{xx} + b(x,t)u_x + c(x,t)u \tag{A.1}$$

where $a(x,t)$ denotes diffusion coefficient, $b(x,t)$ convection coefficient and $c(x,t)$ zero coefficient. Since the coefficients can be functions, each coefficient is implemented as a pure virtual method.

```
virtual double DiffusionCoeff(double t, double x) const = 0;
virtual double ConvectionCoeff(double t, double x) const = 0;
virtual double ZeroCoeff(double t, double x) const = 0;
```

Additionally, initial condition, boundary conditions and analytic solution (if it exists) is implemented in a similar fashion.

```
virtual double BoundaryLeft(double t, double x) const = 0;
virtual double BoundaryRight(double t, double x) const = 0;
virtual double InitCond(double x) const = 0;
virtual double AnalyticSol(double t, double x) const = 0;
```

Appendix B

Implementation of the `FDM` class

The FDM class mainly consists of five pure virtual methods to implement finite different schemes. The class also stores step sizes, coefficients for derivative approximations and solutions at the current and previous time step.

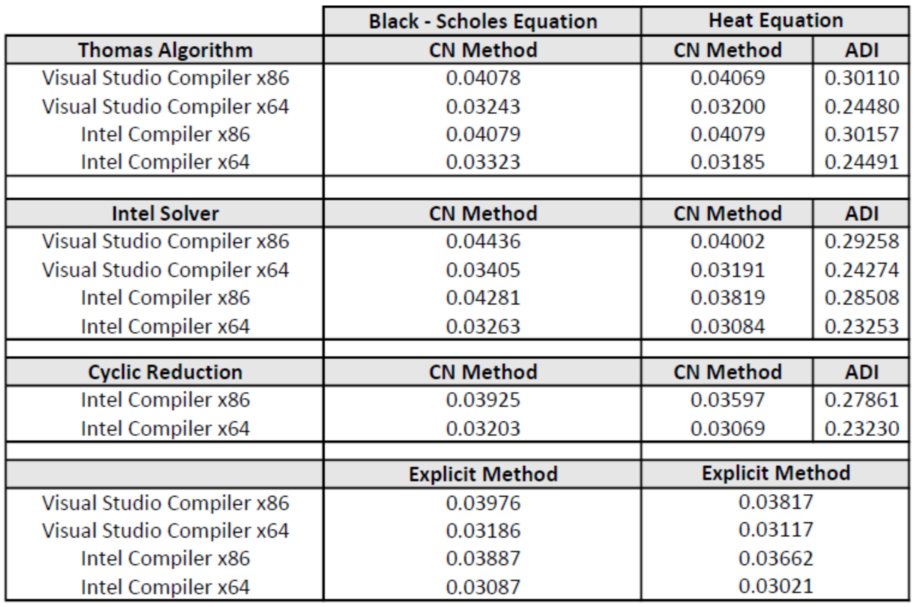
When the class is used, constructor calls `stepSize()` and `initialConditions()`.

- `stepSize()` determines step sizes on space and time dimensions.
- `initialConditions()` resizes the solution vectors with respect to the step size and calls the initial condition from the PDE class to fill the vector with initial values.
- The main difference between schemes is `innerDomain()` method which updates all solution points.
- `boundaryConditions()` calls the boundary condition from the PDE class and fills the boundary values in the vectors.

The user interacts with the solver using `timeMarch()` method which performs the actual looping of time domain. A file stream is opened for a csv file. At each time step`innerDomain()` and `boundaryConditions()` methods are used to calculate the new solution and the result is output to the file.

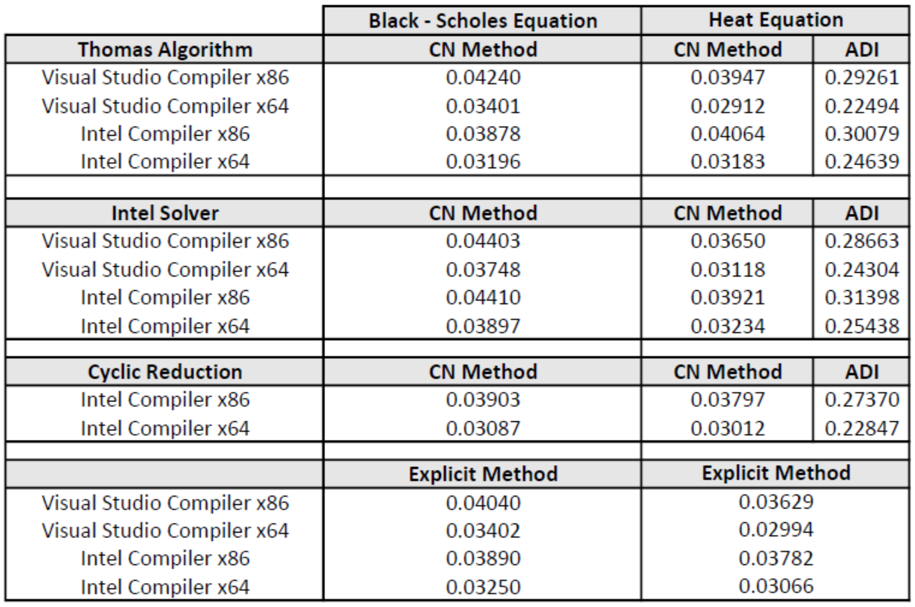
Appendix C

Detailed Timings



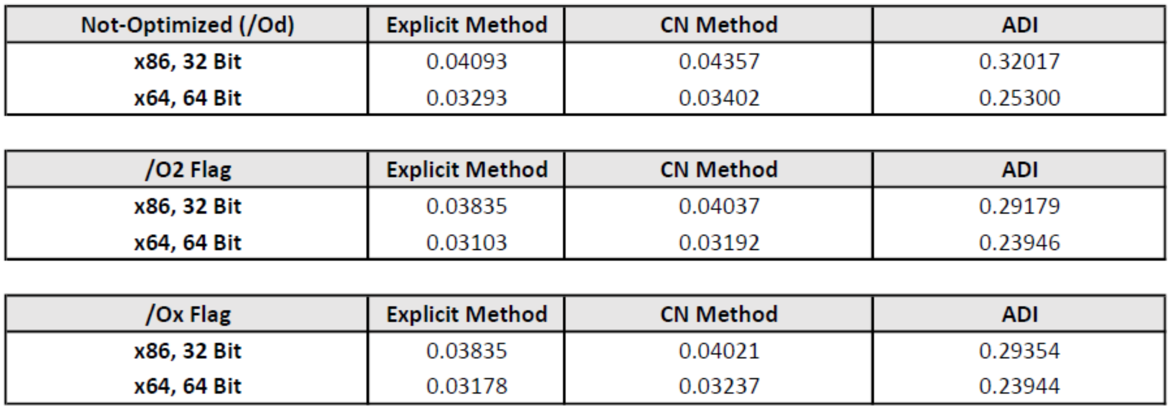
	Black - Scholes Equation	Heat Equation	
Thomas Algorithm	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04078	0.04069	0.30110
Visual Studio Compiler x64	0.03243	0.03200	0.24480
Intel Compiler x86	0.04079	0.04079	0.30157
Intel Compiler x64	0.03323	0.03185	0.24491
Intel Solver	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04436	0.04002	0.29258
Visual Studio Compiler x64	0.03405	0.03191	0.24274
Intel Compiler x86	0.04281	0.03819	0.28508
Intel Compiler x64	0.03263	0.03084	0.23253
Cyclic Reduction	**CN Method**	**CN Method**	**ADI**
Intel Compiler x86	0.03925	0.03597	0.27861
Intel Compiler x64	0.03203	0.03069	0.23230
	Explicit Method	**Explicit Method**	
Visual Studio Compiler x86	0.03976	0.03817	
Visual Studio Compiler x64	0.03186	0.03117	
Intel Compiler x86	0.03887	0.03662	
Intel Compiler x64	0.03087	0.03021	

Figure C.1: Time taken to solve Black - Scholes equation and heat equation using the Explicit, Crank Nicolson and ADI schemes optimized with /O2 flag.



	Black - Scholes Equation	Heat Equation	
Thomas Algorithm	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04240	0.03947	0.29261
Visual Studio Compiler x64	0.03401	0.02912	0.22494
Intel Compiler x86	0.03878	0.04064	0.30079
Intel Compiler x64	0.03196	0.03183	0.24639
Intel Solver	**CN Method**	**CN Method**	**ADI**
Visual Studio Compiler x86	0.04403	0.03650	0.28663
Visual Studio Compiler x64	0.03748	0.03118	0.24304
Intel Compiler x86	0.04410	0.03921	0.31398
Intel Compiler x64	0.03897	0.03234	0.25438
Cyclic Reduction	**CN Method**	**CN Method**	**ADI**
Intel Compiler x86	0.03903	0.03797	0.27370
Intel Compiler x64	0.03087	0.03012	0.22847
	Explicit Method	**Explicit Method**	
Visual Studio Compiler x86	0.04040	0.03629	
Visual Studio Compiler x64	0.03402	0.02994	
Intel Compiler x86	0.03890	0.03782	
Intel Compiler x64	0.03250	0.03066	

Figure C.2: Time taken to solve Black - Scholes equation and heat equation using the Explicit, Crank Nicolson and ADI schemes optimized with /Ox flag.

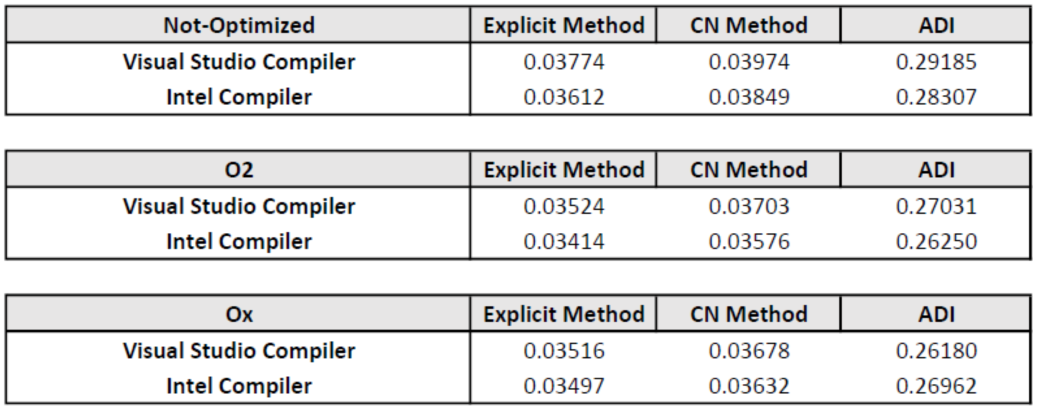


Not-Optimized (/Od)	**Explicit Method**	**CN Method**	**ADI**
x86, 32 Bit	0.04093	0.04357	0.32017
x64, 64 Bit	0.03293	0.03402	0.25300

/O2 Flag	**Explicit Method**	**CN Method**	**ADI**
x86, 32 Bit	0.03835	0.04037	0.29179
x64, 64 Bit	0.03103	0.03192	0.23946

/Ox Flag	**Explicit Method**	**CN Method**	**ADI**
x86, 32 Bit	0.03835	0.04021	0.29354
x64, 64 Bit	0.03178	0.03237	0.23944

Figure C.3: Average increase when switched to 64 bit under different optimization flags.

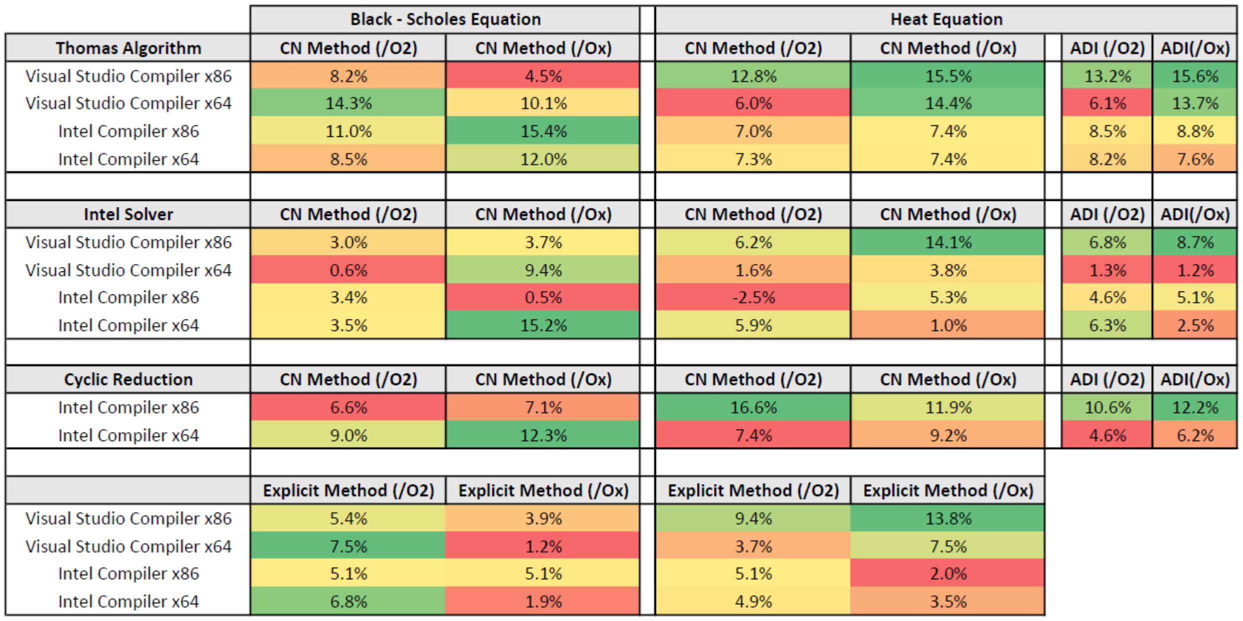


Not-Optimized	Explicit Method	CN Method	ADI
Visual Studio Compiler	0.03774	0.03974	0.29185
Intel Compiler	0.03612	0.03849	0.28307

O2	Explicit Method	CN Method	ADI
Visual Studio Compiler	0.03524	0.03703	0.27031
Intel Compiler	0.03414	0.03576	0.26250

Ox	Explicit Method	CN Method	ADI
Visual Studio Compiler	0.03516	0.03678	0.26180
Intel Compiler	0.03497	0.03632	0.26962

Figure C.4: Average timings comparing Intel Compiler and Visual Studio Compiler.



	Black - Scholes Equation		Heat Equation			
Thomas Algorithm	**CN Method (/O2)**	**CN Method (/Ox)**	**CN Method (/O2)**	**CN Method (/Ox)**	**ADI (/O2)**	**ADI(/Ox)**
Visual Studio Compiler x86	8.2%	4.5%	12.8%	15.5%	13.2%	15.6%
Visual Studio Compiler x64	14.3%	10.1%	6.0%	14.4%	6.1%	13.7%
Intel Compiler x86	11.0%	15.4%	7.0%	7.4%	8.5%	8.8%
Intel Compiler x64	8.5%	12.0%	7.3%	7.4%	8.2%	7.6%
Intel Solver	**CN Method (/O2)**	**CN Method (/Ox)**	**CN Method (/O2)**	**CN Method (/Ox)**	**ADI (/O2)**	**ADI(/Ox)**
Visual Studio Compiler x86	3.0%	3.7%	6.2%	14.1%	6.8%	8.7%
Visual Studio Compiler x64	0.6%	9.4%	1.6%	3.8%	1.3%	1.2%
Intel Compiler x86	3.4%	0.5%	-2.5%	5.3%	4.6%	5.1%
Intel Compiler x64	3.5%	15.2%	5.9%	1.0%	6.3%	2.5%
Cyclic Reduction	**CN Method (/O2)**	**CN Method (/Ox)**	**CN Method (/O2)**	**CN Method (/Ox)**	**ADI (/O2)**	**ADI(/Ox)**
Intel Compiler x86	6.6%	7.1%	16.6%	11.9%	10.6%	12.2%
Intel Compiler x64	9.0%	12.3%	7.4%	9.2%	4.6%	6.2%
	Explicit Method (/O2)	**Explicit Method (/Ox)**	**Explicit Method (/O2)**	**Explicit Method (/Ox)**		
Visual Studio Compiler x86	5.4%	3.9%	9.4%	13.8%		
Visual Studio Compiler x64	7.5%	1.2%	3.7%	7.5%		
Intel Compiler x86	5.1%	5.1%	5.1%	2.0%		
Intel Compiler x64	6.8%	1.9%	4.9%	3.5%		

Figure C.5: Percentage decrease in runtime under the /Ox and /O2 flags using the Explicit, Crank Nicolson and ADI schemes .

Bibliography

[1] Saeed Amen, *Trading thalesians: What the ancient world can teach us about trading today*, pp. 39–60, Palgrave Macmillan UK, London, 2014.

[2] Fischer Black and Myron Scholes, *The pricing of options and corporate liabilities*, Journal of Political Economy **81** (1973), no. 3, 637–654.

[3] Michael J. Brennan and Eduardo S. Schwartz, *Finite difference methods and jump processes arising in the pricing of contingent claims: A synthesis*, The Journal of Financial and Quantitative Analysis **13** (1978), no. 3, 461–474.

[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using openmp: Portable shared memory parallel programming (scientific and engineering computation)*, pp. 23 – 25, The MIT Press, 2007.

[5] Jaehyuk Choi, *Sum of all black–scholes–merton models: An efficient pricing method for spread, basket, and asian options*, Journal of Futures Markets **38** (2018), no. 6, 627–644.

[6] Gustavo Chávez, George Turkiyyah, Stefano Zampini, Hatem Ltaief, and David Keyes, *Accelerated cyclic reduction: A distributed-memory fast solver for structured linear systems*, Parallel Computing **74** (2018), 65 – 83, Parallel Matrix Algorithms and Applications (PMAA'16).

[7] J. Crank and P. Nicolson, *A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type*, Advances in Computational Mathematics **6** (1996), no. 1, 207–226.

[8] Jim Douglas, Jr., *Alternating direction methods for three space variables*, Numer. Math. **4** (1962), no. 1, 41–63.

[9] D.J. Duffy, *Finite difference methods in financial engineering: A partial differential equation approach*, The Wiley Finance Series, Wiley, 2006.

[10] ______, *Financial instrument pricing using c++*, The Wiley Finance Series, Wiley, 2013.

[11] G. Evans, J.M. Blackledge, and P. Yardley, *Numerical methods for partial differential equations*, Springer undergraduate mathematics series, Springer, 2000.

[12] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein, *From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers*, 2019 USENIX Annual Technical Conference (USENIX ATC 19) (Renton, WA), USENIX Association, July 2019, pp. 475–488.

[13] P. Glasserman, *Monte carlo methods in financial engineering*, Applications of mathematics : stochastic modelling and applied probability, pp. 2 − 3, Springer, 2004.

[14] R. W. Hockney, *A fast direct solution of poisson's equation using fourier analysis*, J. ACM **12** (1965), no. 1, 95–113.

[15] John Hull, *Options, futures, and other derivatives*, ninth edition ed., Pearson Education Limited, 2018.

[16] Intel, *Intel math kernel library for c, gtsv*, https://software.intel.com/en-us/mkl-developer-reference-c-gtsv, 2019, [Online; accessed 7-August-2019].

[17] Stephen Jewson and Mihail Zervos, *The black-scholes equation for weather derivatives*, SSRN Electronic Journal (2003).

[18] F.C. Klebaner, *Introduction to stochastic calculus with applications*, Introduction to Stochastic Calculus with Applications, Imperial College Press, 2005.

[19] P. Kohl-Landgraf, *Pde valuation of interest rate derivatives: From theory to implementation*, pp. 423–438, Books on Demand, 2007.

[20] Daniel Kusswurm, *Modern x86 assembly language programming: 32-bit, 64-bit, sse, and avx*, 1st ed., pp. 2–3, Apress, Berkely, CA, USA, 2014.

[21] Matthew J. Hancock, *The 1-d heat equation (mit course 18.303, linear partial differential equations)*, https://ocw.mit.edu/courses/mathematics/18-303-linear-partial-differential-equations-fall-2006/lecture-notes/heateqni.pdf, 2006, [Online; accessed 7-August-2019].

[22] Robert C. Merton, *Option pricing when underlying stock returns are discontinuous*, Journal of Financial Economics **3** (1976), no. 1, 125 – 144.

[23] Microsoft, */gf (eliminate duplicate strings)*, https://docs.microsoft.com/en-us/cpp/build/reference/gf-eliminate-duplicate-strings?view=vs-2019, 2016, [Online; accessed 10-August-2019].

[24] ———, */gy (enable function-level linking)*, https://docs.microsoft.com/en-us/cpp/build/reference/gy-enable-function-level-linking?view=vs-2019, 2016, [Online; accessed 10-August-2019].

[25] ———, */ob (inline function expansion)*, https://docs.microsoft.com/en-us/cpp/build/reference/ob-inline-function-expansion?view=vs-2019, 2016, [Online; accessed 10-August-2019].

[26] ———, */oi (generate intrinsic functions)*, https://docs.microsoft.com/en-us/cpp/build/reference/oi-generate-intrinsic-functions?view=vs-2019, 2016, [Online; accessed10-August-2019].

[27] ———, */os, /ot (favor small code, favor fast code)*, https://docs.microsoft.com/en-us/cpp/build/reference/os-ot-favor-small-code-favor-fast-code?view=vs-2019, 2016, [Online; accessed 10-August-2019].

[28] ———, */o options (optimize code)*, https://docs.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=vs-2019, 2017, [Online; accessed 10-August-2019].

[29] ———, */og (global optimizations)*, https://docs.microsoft.com/en-us/cpp/build/reference/og-global-optimizations?view=vs-2019, 2017, [Online; accessed 10-August-2019].

[30] ———, */oy (frame-pointer omission)*, https://docs.microsoft.com/en-us/cpp/build/reference/oy-frame-pointer-omission?view=vs-2019, 2018, [Online; accessed 10-August-2019].

[31] K. W. Morton and D. F. Mayers, *Numerical solution of partial differential equations: An introduction*, 2 ed., Cambridge University Press, 2005.

[32] Martin Neuenhofen, *A time-optimal algorithm for solving (block-) tridiagonal linear systems of dimension n on a distributed computer of n nodes*, arXiv preprint arXiv:1801.09840 (2018).

[33] The Royal Swedish Academy of Sciences, *The sveriges riksbank prize in economic sciences in memory of alfred nobel 1997*, https://www.nobelprize.org/prizes/economic-sciences/1997/press-release/, Oct 1997, [Online; accessed 28-August-2019].

[34] Samuel Palmer, *Accelerating implicit finite difference schemes using a hardware optimised implementation of the thomas algorithm for fpgas*, arXiv preprint arXiv: 1402.5094, 2014 (2014).

[35] D. W. Peaceman and H. H. Rachford, *The numerical solution of parabolic and elliptic differential equations*, Journal of the Society for Industrial and Applied Mathematics **3** (1955), no. 1, 28–41.

[36] Charles Petzold, *Programming windows, fifth edition*, 5th ed., Microsoft Press, Redmond, WA, USA, 1998.

[37] Pablo Quesada-Barriuso, Julián Lamas-Rodríguez, Dora B Heras, Montserrat Bóo, and Francisco Argüello, *Selecting the best tridiagonal system solver projected on multi-core cpu and gpu platforms*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), The Steering Committee of The World Congress in Computer Science, Computer . . . , 2011, p. 1.

[38] Sriram Ramaswamy, *Pollen grains, random walks and einstein*, Resonance **5** (2000), 106–124.

[39] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*, pp. 19–20, Clarendon Press, 1985.

[40] D. Tavella and C. Randall, *Pricing financial instruments: The finite difference method*, Wiley Series in Financial Engineering, Wiley, 2000.

[41] Andrew V Terekhov, *Parallel dichotomy algorithm for solving tridiagonal system of linear equations with multiple right-hand sides*, Parallel Computing **36** (2010), no. 8, 423–438.

[42] The OpenMP Architecture Review Board, *What problem does openmp solve?*, `https://www.openmp.org/about/openmp-faq/`, 2018, [Online; accessed 10-August-2019].

[43] J.W. Thomas, *Numerical partial differential equations: Finite difference methods*, Texts in Applied Mathematics, Springer New York, 2013.

[44] Llewellyn Hilleth Thomas, *Elliptic problems in linear difference equations over a network*, Watson Sci. Comput. Lab. Rept., Columbia University, New York **1** (1949).

[45] Linda Torczon and Keith Cooper, *Engineering a compiler*, 2nd ed., pp. 19–21, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[46] University of Tennessee, *Lapack: Linear algebra package, dgtsv()*, `http://www.netlib.org/lapack/explore-html/d4/d62/group__double_g_tsolve_ga2bf93f2ddefa5e671866eb2191dc19d4.html#ga2bf93f2ddefa5e671866eb2191dc19d4`, 2016, [Online; accessed 17-August-2019].

[47] Ruud van der Pas, Eric Stotzer, and Christian Terboven, *Using openmp – the next step: Affinity, accelerators, tasking, and simd*, 1st ed., pp. 25 – 28, The MIT Press, 2017.

[48] Paul Wilmott, *Paul wilmott introduces quantitative finance*, 2 ed., Wiley-Interscience, New York, NY, USA, 2007.

[49] Tomasz Zastawniak and Maciej J. Capinski, *Numerical methods in finance with c++*, Mastering Mathematical Finance, pp. 149 – 150, Cambridge University Press, 7 2012 (English).

[50] Yao Zhang, Jonathan Cohen, and John D. Owens, *Fast tridiagonal solvers on the gpu*, SIGPLAN Not. **45** (2010), no. 5, 127–136.