Generative Programming Methods for Parallel Partial Differential Field Equation Solvers

A thesis presented in partial fulfilment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Albany
New Zealand

Daniel Peter Playne

2011

Abstract

This thesis describes a generative programming system that automatically constructs parallel simulations of complex systems that are based on field equations using finite differencing and explicit Runge-Kutta integration methods. Programming computational simulations by hand for different parallel architectures is both tedious and time consuming. Simulation frameworks struggle to target different architectures without losing performance. Automating the process of constructing simulation codes can significantly improve productivity.

Three computational models based on field equations are discussed in depth along with numerical methods for discretising and simulating them. The Cahn-Hilliard model of phase separation, the Ginzburg-Landau model of superconductivity and the Lotka-Volterra model of interacting populations are discussed in detail and are used as examples. A number of modern parallel computing architectures and associated parallel programming languages are also discussed and simulation implementations that run upon these architectures are presented. The performance results from these implementations are used to compare the parallel architectures and their relative performance capabilities for processing each type of simulation.

The key elements of a simulation are identified as being: the computational model, the stencil operators, the explicit integration methods and the configuration information. A domain specific language for defining these elements is developed and presented. A generative programming system is described that parses these element definitions and combines them together to form a complete simulation definition. It is then shown that code generators can be readily developed to produce very efficient implementations from these simulation definitions in any parallel programming language including C, TBB, MPI and CUDA for which explicit examples are given. It is shown that this method is effective, extensible and can considerably reduce the programmer effort required to develop fast parallel simulations of complex systems.

Acknowledgements

First of all I should like to thank Ken Hawick and Chris Scogings for their wisdom, patience and encouragement throughout this work. My deepest thanks to my parents David and Cheryl, my brother Matthew and Rebecca Pearman for their continual love and support. Also to my colleagues Arno Leist, Anton Gerdelan, Teo Susnjak and Guy Kloss for their advice and many discussions. I wish to acknowledge The North Harbour Club, Massey University and the New Zealand Tertiary Education Commission for their financial support. Finally I would like to thank everyone who has supported and contributed to making this work possible.

Contents

1 Introduction **1**
- 1.1 The Challenge of Simulation Development . . . 1
- 1.2 Introduction to Complex Systems . . . 1
- 1.3 Introduction to Field Equations . . . 2
- 1.4 Introduction to Parallel Computing . . . 3
- 1.5 Introduction to Automatic Parallelization . . . 4
- 1.6 Introduction to Generative Programming . . . 5
- 1.7 Previous Work . . . 6
- 1.8 Aim of this Thesis . . . 7
- 1.9 Thesis Structure . . . 8

2 Models **9**
- 2.1 Introduction . . . 9
- 2.2 Cahn-Hilliard Equation . . . 10
- 2.3 Time Dependent Ginzburg-Landau Equation . . . 11
- 2.4 Lotka-Volterra Equation . . . 14
- 2.5 Model Commonalities . . . 18
- 2.6 Conclusions . . . 18

3 Numerical Methods **19**
- 3.1 Introduction . . . 19
- 3.2 Finite-Differencing . . . 20
- 3.3 Boundary Conditions . . . 22
- 3.4 Time Integration Methods . . . 25
- 3.5 Butcher Tableaux . . . 27
- 3.6 Commonalities in Numerical Methods . . . 28
- 3.7 Higher-Order Runge-Kutta Methods . . . 29
- 3.8 Conclusions . . . 30

4 Parallel Architectures and Languages **31**
- 4.1 Introduction . . . 31
- 4.2 Central Processing Units . . . 32
- 4.3 Cluster Computers . . . 34
- 4.4 Graphical Processing Units . . . 36
- 4.5 Multi-GPU . . . 41
- 4.6 GPU-Accelerated Clusters . . . 42
- 4.7 Commonalities in Parallel Architectures . . . 44
- 4.8 Conclusions . . . 44

5 Parallel Algorithms **45**
- 5.1 Introduction . . . 45
- 5.2 Equation Computation . . . 46
- 5.3 Boundary Conditions . . . 49
- 5.4 Sequential Implementation . . . 50
- 5.5 Parallel Decomposition . . . 53
- 5.6 Multi-Core - POSIX Threads . . . 54
- 5.7 Multi-Core -Threading Building Blocks . . . 56
- 5.8 Cluster - MPI . . . 57
- 5.9 Graphical Processing Units - CUDA . . . 59

5.10 Multi-GPU - CUDA & Pthreads . . . 63
5.11 GPU Cluster - CUDA & MPI . . . 67
5.12 Simulation Commonalities . . . 68
5.13 Conclusions . . . 69

6 Performance Results **71**
6.1 Introduction . . . 71
6.2 Model Computation . . . 71
6.3 Single-Core CPU . . . 72
6.4 Multi-Core CPU . . . 73
6.5 Cluster . . . 76
6.6 Tesla GPU . . . 77
6.7 Fermi GPU . . . 81
6.8 Multi-GPU . . . 83
6.9 GPU Cluster . . . 83
6.10 Performance Comparison . . . 84
6.11 Conclusions . . . 85

7 Simulation Description Language **87**
7.1 Introduction . . . 87
7.2 Domain Specific Language . . . 88
7.3 Equation Parser . . . 89
7.4 Stencil Operators . . . 93
7.5 Integration Methods . . . 95
7.6 Configuration . . . 97
7.7 Conclusions . . . 98

8 Simulation Representation **99**
8.1 Introduction . . . 99
8.2 Equation Trees . . . 100
8.3 Stencil Nodes . . . 102
8.4 Integration Trees . . . 105
8.5 Merging Equation and Integration Trees . . . 108
8.6 Conclusions . . . 114

9 Automatic Code Generation **117**
9.1 Introduction . . . 117
9.2 Equation Generation . . . 118
9.3 Generating Target Specific Code . . . 121
9.4 Generating C code . . . 121
9.5 Generating TBB code . . . 124
9.6 Generating MPI code . . . 127
9.7 Generating CUDA code . . . 131
9.8 STARGATES Results . . . 135
9.9 Conclusions . . . 136

10 Conclusions **137**
10.1 Thesis Overview . . . 137
10.2 Major Contributions . . . 139
10.3 Implications for Automatic Parallelism . . . 140
10.4 Implications for Computational Simulations . . . 141
10.5 Future Work . . . 142

A Adaptive Stepsize Methods **147**
A.1 Introduction to Adaptive Stepsize Methods 147
A.2 Runge-Kutta Methods with Adaptive Stepsizes 147

B Generated Code Example **151**

List of Figures

2.1 Three-dimensional Cahn-Hilliard visualisations . . . 11
2.2 The three quenching phases of the Cahn-Hilliard model . . . 12
2.3 Coarsening behaviour of the Cahn-Hilliard model . . . 13
2.4 Two-dimensional visualisation methods for the Ginzburg-Landau equation . . . 14
2.5 Three-dimensional Ginzburg-Landau visualisations . . . 15
2.6 Three-dimensional Lotka-Volterra visualisations . . . 16
2.7 Lotka-Volterra Species Populations . . . 17

3.1 The Laplacian operator . . . 21
3.2 The Biharmonic operator . . . 22
3.3 Periodic boundary conditions . . . 23
3.4 Dirichlet boundary conditions . . . 24
3.5 Neumann boundary conditions . . . 25

4.1 Single-Core CPU architecture . . . 32
4.2 Multi-Core CPU architecture . . . 33
4.3 Cluster computer architecture . . . 35
4.4 Tesla GPU architecture . . . 37
4.5 Fermi GPU architecture . . . 39
4.6 Multi-GPU architecture . . . 42
4.7 GPU cluster architecture . . . 43

5.1 Two-dimensional lattice decomposition . . . 53
5.2 Three-dimensional lattice decomposition . . . 54

6.1 Cahn-Hillard, Ginzburg-Landau and Lotka-Volterra processing time on a CPU . . . 72
6.2 Cahn-Hillard, Ginzburg-Landau and Lotka-Volterra processing time on a GPU . . . 73
6.3 Comparison of integration methods on a single CPU core . . . 74
6.4 Comparison of thread numbers for Pthreads implementation . . . 75
6.5 Comparison of Pthreads and single-core implementations . . . 75
6.6 Performance of different TBB task block sizes . . . 76
6.7 Comparison of the TBB and single-core implementation . . . 77
6.8 Comparison of MPI nodes . . . 78
6.9 Comparison of the MPI and single-thread implementation . . . 78
6.10 Comparison of thread block sizes for Tesla GPUs . . . 79
6.11 Comparison of memory types for Tesla GPUs . . . 80
6.12 Comparison of the Tesla and single-core implementations . . . 80
6.13 Comparison of thread block sizes for Fermi GPUs . . . 81
6.14 Comparison of memory types for Fermi GPUs . . . 82
6.15 Comparison of the Fermi and single-core implementations . . . 82
6.16 Comparison of the Multi-GPU and single-GPU implementations . . . 83
6.17 Comparison of all implementations . . . 84

7.1 Laplacian, Biharmonic and SpatialAverage stencils . . . 94
7.2 Extension of Laplace stencil to higher dimensions . . . 95

8.1 Cahn-Hilliard equation tree . . . 101
8.2 Ginzburg-Landau equation tree . . . 101
8.3 Lotka-Volterra equation tree . . . 102

8.4 Stencil operator arrays 103
8.5 Rearranged Cahn-Hilliard equation tree 104
8.6 Application of two stencil operators 105
8.7 Euler method integration tree 107
8.8 RK2 method integration tree 108
8.9 Cahn-Hilliard Euler simulation tree 111
8.10 Cahn-Hilliard RK2 simulation tree 112
8.11 Lotka-Volterra RK2 simulation tree 115

List of Tables

3.1 General form of a Butcher Tableau 28
3.2 Euler method Butcher tableau 28
3.3 RK4 method Butcher tableau 28
3.4 RK4(2) method Butcher tableau 29
3.5 RK5 method Butcher tableau 30
3.6 RK6 method Butcher tableau 30

6.1 Comparison of parallel architectures and models 85

7.1 General form of a Butcher tableau 95

A.1 Merson method Butcher tableau 148
A.2 Fehlberg $5^{th}/6^{th}$ method Butcher tableau 149
A.3 Verner $5^{th}/6^{th}$ method Butcher tableau 150
A.4 Dormand-Prince $4^{th}/5^{th}$ method Butcher tableau 150

"The beginning is the most important part of the work."

Plato

1 Introduction

1.1 The Challenge of Simulation Development

Developing simulations of complex systems for parallel machines is a challenging task. Simulation codes are becoming increasingly intricate and use complex numerical methods. In addition to this parallel architectures and languages are continually changing and evolving. The evolution of these architectures and languages compounds the challenge of maintaining simulations and migrating them to the latest machines.

An improved method of defining simulations and deploying them to parallel machines is required. Simulations frameworks have some success by creating additional software layers which can be implemented for any new machines. Unfortunately these frameworks tend to introduce inefficiency into solutions and struggle to make full use of the different parallel architectures.

Generative programming is presented as an alternative to these frameworks. This type of system requires the elements of a simulation to be defined in a regular way that can be interpreted and combined to produce a target solution. The advantage of these systems is that the simulation description remains the same, regardless of the target architecture. The implementations produced by these systems are lightweight and architecture specific but the system can easily be configured to produce new implementations for other architectures.

1.2 Introduction to Complex Systems

The term complexity is used by many different disciplines to represent a vast range of different concepts. This thesis considers the complexity of behaviour emerging from complex systems and computational models [1, 2]. It should be clear that this does not refer to computational complexity which considers the cost of computing a model. It considers instead the complexity of the model's dynamics. This field of research explores the complex behaviour that emerges from computational models. Models that are governed by a simple set of rules or equations can exhibit amazing and irregular patterns and behaviour. The typical example is Wolfram's Cellular Automata [1] which is an extremely simple model that produces highly irregular behaviour in certain conditions.

Simple abstract models such as these often exhibit very similar behaviour and dynamics to natural systems. This comparison can be made by comparing model behaviour to real-world systems [1] or more commonly by comparing them to models designed to approximate these systems. Adopting the Principle of Computational Equivalence allows all natural processes to be considered as computations. This allows all computational models to be considered together regardless of whether they model natural systems or are abstract constructions.

This allows many different computational models from a whole range of origins - physics, chemistry, biology, mathematics or the entirely abstract to be compared. As yet there is no accepted theory of complexity and the underlying mechanisms of these complex systems remain unknown [2]. The long-term goal for the study of complexity is to develop methods of characterising complex systems to allow them to be classified. This could lead to the development of a single unified theory of complexity.

A unified theory of complexity could have untold benefits and implications for our understanding and thinking about the world. The complex behaviour of turbulent fluids is considered a great scientific challenge [3]. In the words of Richard Feynman turbulence is "the most important unsolved problem of classical physics" [3]. Any theory of complexity that could even contribute to this (and many other fields) would be invaluable.

There exists a huge number of complex systems and computational models and exploring the properties, dynamics and behaviour of all of these models is far too much work to address within a single thesis. Instead of a detailed exploration of these models, methods that aid in the investigation and comparison of these models are developed. Even this task presents a large amount of work so a subset of the possible computational models has been identified and investigated - namely field equations.

1.3 Introduction to Field Equations

Field equations are one family of computational models which describe the behaviour of a field of interacting matter or entities. Many different field equations have been described by a great number of different authors to describe systems across a range of different research fields. Systems that can be modelled using field equations include - Binary Alloys [4, 5], Electromagnetism [6], Ferromagnetism [7, 8], Fluid Dynamics [9–11], Gravitational Fields [12], Quantum Dynamics [13], Relativity [14, 15], Species Populations [16, 17], Superconductivity [18, 19], Thermodynamics [20] and Weather Systems [21, 22].

These equations usually model the system as a continuous field, however numerical methods can be used to discretise the fields and find approximate solutions. These numerical methods allow solutions to these systems to be approximated by computer simulations. These computer simulations can offer insight into the systems they model. This can take the form of classifying the spatial patterns or behaviour of the simulations or measuring specific properties of the systems computed. Such simulations have a wide range of applications.

Simulations approximating a real-world system can be computed in faster than real-time to predict the future state of the real-world system. This is commonly used in weather forecasting systems

which approximate the many complex interactions in the atmosphere to predict the weather in future days. Such simulations can also be used to calculate the potential of gravitational fields, allowing the motion of planets, asteroids, satellites and the like to be predicted.

Other simulations can be used to search model parameters to identify the parameter and initial conditions that result in a system with a desired set of properties. This can be useful for models such as models of quenching binary alloys. Exploration of such a model's parameter space can identify quenching methods that can produce alloys with the strongest structures. Fluid dynamics simulations are commonly used in the design of cars, planes, ships and buildings. These simulations allow a design to be evaluated in terms of aerodynamic efficiency.

These models can also represent purely abstract or mathematical constructs which do not relate to any real-world system. The study of such models is still an interesting field which examines their emergent behaviour. Simple abstract models can often exhibit complex behaviour very similar to that of detailed models approximating real-world phenomena.

1.4 Introduction to Parallel Computing

Parallel computing is a wide area of research which seeks to increase computational performance by using multiple cores, processors or machines to collaborate on a single task [23]. This operates on the principle that many computational tasks can be split into separate problems that can be computed independently. Parallel programs can range in scale from a simple multi-threaded program executing on a single processor to distributed programs executing on hundreds or thousands of machines.

Parallel machines first emerged in the early 1960s [24]. Interest in parallel machines grew in the 1970s with the development of Single Instruction Multiple Data (SIMD) machines such as the Illiac IV [25], the Distributed Array Processor [26] and the Massively Parallel Processor [27]. More widespread interest developed in parallel computing with the release of Multiple Instruction Multiple Data (MIMD) machines such as the Denelcor HEP [28] and the CRAY X-MP/22 [29]. Modern supercomputers have evolved into massively parallel machines containing very large numbers of networked machines. On the June 2011 TOP500 list, the fastest supercomputer contains over half a million processors [30].

Early distributed parallel computers also started emerging during this period. Most modern parallel computer developments are conceptually similar to these parallel machines. There are many different processing architectures currently available which can be used to execute parallel programs. Most modern desktop machines contain central processing units with at least two cores but CPUs with up to four or six are not uncommon. Motherboards that are capable of hosting multiple CPUs are also available. To allow multiple CPU cores to collaborate on a single task, thread synchronisation and mutual exclusion must be used to ensure the computation is performed correctly. These cores can communicate information through the memory of the machine.

Graphical processing units common in many household computers have evolved into highly parallel and computationally powerful processing architectures capable of rendering modern three-dimensional graphics. These graphics cards can be used for general computing tasks, known as General Purpose Computation on Graphical Processing Units or GPGPU [31–33]. These GPUs can

provide a high level of parallelisation and computational power for a relatively low cost. This relatively new parallel architecture relies on many low-power cores to provide high overall throughput.

Most large scale supercomputers are based on a distributed set of compute nodes connected by a network; such machines can range from small research clusters to international grids or cloud machines. These distributed memory machines must not only split computation between nodes but also the data storage. The main challenge for programming these machines is managing the required communication between the nodes. These distributed compute nodes may contain multiple core CPUs, multiple processors or other accelerators such as GPUs.

Parallel computing has allowed computational problems to be solved that would have been otherwise impossible with the computing hardware available at the time. With the physical limitations chip manufacturers are encountering at the present time, parallel computing is more important than ever. Chip designers are increasing the number of parallel cores to continue increasing the computational performance of central processing units. This increase in widespread parallelism makes the development of parallel programs an ever more important research area.

1.5 Introduction to Automatic Parallelization

Automatic Parallelization is considered one of the grand challenges of Computer Science [34, 35]. This area of research considers how sequential code can be automatically converted to make use of multiple processing cores. The goal of automatically parallelising compilers is to analyse any sequential program and determine any dependencies and if/how the program can be safely parallelised. Such compilers take this complex and error-prone responsibility away from the user.

Exploiting parallelism at any level is important for applications to make use of the parallel processing cores in modern processor architectures. Parallelism can be implemented and utilised at several different levels from instruction-level parallelism and data-parallelism to task-parallelism [34]. The compiler must be capable of analysing a program to determine the dependencies that exist within it and which instructions can be safely executed in parallel. Analysis can be performed statically at compile-time or also at run-time [34]. This dependence analysis is a very complicated process which must be able to correctly determine all dependencies in any program which may include indirect addressing, recursion, pointers and other such program features [36,37]. Automatically parallelising compilers must never introduce any unsafe parallelism.

Because instructions within loops are executed many times during a single program execution, they represent the largest computational portion of most programs. A significant reduction in execution time can be achieved if these loop instructions can be effectively parallelised. Most automatically parallelising compilers will focus on how loop instructions can be executed in parallel.

Such compilers must not only determine if instructions can be parallelised but also whether it is worth parallelising them [35]. Executing instructions in parallel often requires a certain overhead which can outweigh the benefits gained by parallel execution. Compilers that introduce unnecessary and inefficient parallelism can increase the execution time of an application - the opposite result to the one intended.

More success has been achieved by languages that allow the user to give 'hints' or extra infor-

mation to the compiler about how a section of code may be parallelised. One of the most significant languages of this type is High Performance Fortran or HPF [38]. HPF is a standardisation of several existing languages such as Fortran D [39], Vienna Fortran [40] and CM Fortran [37, 41].HPF allows users to include directives into a program which can be used by the compiler to parallelise the code. All parallelisation and communication is generated automatically by the compiler and makes it significantly easier for the user to write code that can be executed in parallel.

High Performance Fortran did not achieve the success that was hoped for - it struggled with inconsistent implementations, lack of support and poor performance [37]. However with improvements to compiler technology, additional knowledge about parallel languages and the requirement for improved parallel languages, interest in HPF and such languages is again growing [37].

1.6 Introduction to Generative Programming

Generative programming is an entire paradigm that considers program families rather than individual software solutions [42–44]. The paradigm considers the commonalities and variations of specific solutions within a family and how to describe them in a systematic way [44]. This systematic description of a particular solution allows the software solution to be generated automatically using a number of common components.

Generative programming relies on identification of a family of program solutions which all have similar components but differ in terms of combinations. The family of solutions defines the problem space upon which the solutions are built. It also requires a certain amount of configuration knowledge such as combination rules and optimisations that allow different specific solutions to be defined. Finally a method of converting this domain and configuration information into a solution implementation is required [42]. This paradigm encompasses many areas of research including Domain Specific Languages (DSLs), Feature Modelling, Software Architectures, Code Generators etcetera.

Generative programming is often likened to an assembly line [42, 45]. The assembly line only produces one type of product, such as a car, but is capable of producing cars with many different configurable parts (body, engine, airbags, air conditioning etc) [45]. The desired product is described in an abstract, high-level way and is then produced by the assembly line. This requires a certain knowledge of how the different components can be combined and what combinations are valid/invalid.

Generative programming aims to emulate this concept for software development. Rather than constructing each particular software solution by hand, as is commonly the case, the desired solution is described in a high-level abstract way. This high-level abstract description is then automatically transformed into a specific implementation of the solution by connecting the appropriate software components [45]. This generation phase requires specific knowledge of the target language and architecture to determine how the components can be combined and what optimisations are valid.

Such GP systems can automatically parallelise solutions in a very different way to most automatically parallelising compilers. Rather than introducing parallelism by analysing an arbitrary sequential program and determining which instructions can be executed in parallel, a GP system is

designed for a specific program family and has knowledge of the problem and solutions. This higher level of information allows a GP system to introduce parallelism into the solutions without the need for the same level of dependence analysis. This can be in the form of instruction-level parallelism, data-parallelism or task-parallelism. It can also change the nature of the parallel solutions produced based on the target architecture for which they are intended.

If a Generative Programming software assembly line can be constructed to produce simulation implementations for computational models it can be of great use for exploring and comparing different models. A GP system that allows computational models and the numerical methods used to simulate them at a high level and produce efficient parallel simulations would be of great value for investigating the complexity of computational models. Automating the process of constructing and parallelising simulations can greatly reduce the development time for simulations and allows a greater number of different models to be compared. It can also aid in the maintenance and management of simulations as high-level descriptions and solution mappings are maintained as opposed to individual implementations.

To construct a Generative Programming assembly line for parallel simulations, a Domain Specific Language must be developed to allow the models to be defined in a high-level abstract way. To produce simulation implementations from these high-level descriptions, the commonalities between the implementations must be identified to create a mapping from the problem domain to the target solution domain.

There is generally an overhead involved in initially developing this automated process but it can provide important long-term benefits. The full range of possible solution combinations can be easily explored due to their automated creation. New components can be easily introduced into the process which makes them available to the entire program family.

1.7 Previous Work

There is already a great body of research into complex systems, field-equations, numerical methods, parallel computing and their combination as parallel simulations. Complexity and complex systems have been discussed in depth by a number of authors [2, 46, 47]. A particularly detailed discussion of complexity, and in particular the Principle of Computational Equivalence, is presented in Stephen Wolfram's "A New Kind of Science" [1]. But as yet a complete theory of complexity is a long way off.

Many years of research have been dedicated to producing computational simulations for a wide range of systems. Many different simulations and methods for computing them have been aptly described by various authors [48–50]. Some particularly relevant simulations with parallel implementations include simulations of electromagnetic dynamics [51–53], binary alloys [5], fluid dynamics [54], ice sheets [55]. Simulations of these systems have been implemented for a variety of different parallel architectures and libraries: GPUs and CUDA [53, 56], distributed machines using MPI [51, 57–59], or a combination of multiple parallel devices [54].

The numerical methods used by these simulations have a very long history. Some important works include work by Carl Runge and Wilhelm Kutta [60, 61]. These methods are very well de-

scribed in John Butcher's "Numerical Methods for Ordinary Differential Equations" [62].

Parallel computing is a widely-researched research area with contributions from thousands of authors. This research encompasses the design and construction of parallel machines, network design, operating systems, communication protocols, applications development etcetera. It is worth noting the work of two authors in particular - Geoffrey Fox and his part in the development of the Caltech Cosmic Cube and his famous book "Parallel Computing Works!" [23] and Jack Dongarra and his work developing BLAS, LINPACK and LAPACK [63–66].

Basic Linear Algebra Subprograms or BLAS is a programming interface for performing basic linear algebra operations [63]. The basic linear algebra building blocks allow more complex linear algebra libraries such as LINPACK [64] and the Linear Algebra PACKage (LAPACK) [66] to be constructed. The Scalable Linear Algebra PACKage (ScaLAPACK) is a distributed memory version of this package built on top of Parallel BLAS (PBLAS). These linear algebra packages are similar in nature to the type of parallel computation required by the simulations considered in this work. However as these packages are intended for general linear algebra operations, they tend to be slower than explicitly-written implementations.

Less research has been conducted in the area of generative programming methods and their application to computational simulations. Much of the research into generative programming has been driven by Krzysztof Czarnecki [42, 67]. Some generation systems for partial differential equation finite-difference solvers do exist and include electromagnetism simulation generators [68] and environmental modelling [69]. These generators tend to be focused towards generating sequential FORTRAN and C [68,70,71] although some are designed to produce parallel code [70,72].

The DOLFIN [73] and FEniCS [74] projects are related work which aim to automate the process of developing finite-element variational formulations of partial/ordinary differential equation solvers and linear algebra. DOLFIN provides an interface to FEniCS and relies on code generation to reduce coupling between components. The code generated from the problem definition can then be combined into existing finite element environment. FEniCS relies on a number of linear algebra library back ends.

1.8 Aim of this Thesis

This thesis aims to show that generative programming methods can be applied to computational model simulations to automatically produce implementations for multiple platforms without significant loss of performance.

One family of computational models - field equations - are investigated in particular with the aim of determining how they can be described in a high-level way. Several models are investigated to gain a greater understanding of the models to determine what elements are common features of the program family and which are individual to the model. Determining these differences and commonalities is vital to developing a language to describe them.

A similar investigation is undertaken for the numerical models used to approximate solutions to these models. The aim is the same, to determine how they can be described in an abstract and simple way.

Several modern parallel computing languages and architectures are evaluated to compare their performance for processing lattice-based computational models. Various methods for decomposing these simulations into parallel tasks and implementing them on the different architectures are described and discussed. This process aims to identify the common features of these implementations to determine how simulations can be automatically deployed to these architectures.

These investigations aim to identify how this family of computational models can be described in an abstract way and program components be assembled automatically to produce parallel simulations. This will require the creation of a domain specific language to allow different computational models and the numerical methods to be described at a high level. Parallel components and target specific code generators must be constructed to automatically produce parallel simulation code from these high-level descriptions.

1.9 Thesis Structure

This thesis should read in a linear way with each chapter building on the last. Chapter 2 discusses the partial differential field equations considered in this thesis and introduces the three example models used in this work, the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra models. Chapter 3 presents the numerical methods that can be used to approximate solutions of these models and simulate their behaviour.

In Chapter 4 a number of modern parallel computing architectures and languages are discussed and compared. Parallel implementations of the three example models from Chapter 2 are presented and compared in Chapter 5 using the numerical methods from Chapter 3. The performance of these parallel implementations are presented and compared in Chapter 6.

The next three chapters discuss the generative programming system developed to construct these parallel implementations automatically. Chapter 7 discusses the elements of a simulation and the Domain Specific Language developed to describe them. The process of internally storing these elements and combining them together to form an abstract representation of a simulation is discussed in Chapter 8. Several example code generators that extract information from the abstract simulation representations and construct parallel implementations for various target architectures are discussed in Chapter 9.

An overview of the work and discussion of their implications is presented in Chapter 10 along with some final conclusions and potential future work.

"The universe, they said, depended for its operation on the balance of four forces which they identified as charm, persuasion, uncertainty and bloody-mindedness."

Terry Pratchett

2 Models

2.1 Introduction

Many physical and mathematical systems can be described by a partial differential field equation or by a set of these equations. This particular type of equation models the behaviour and interactions of matter in a continuous field. Such equations can be formulated to model the dynamics of many different types of systems. Examples of such systems include: Binary Alloys [4, 5], Electromagnetic dynamics [6], Ferromagnetism [7, 8], Fluid Dynamics [9–11], Predator-Prey populations [16, 17], Quantum Mechanics [13], Relativity [14, 15], Superconductivity [18, 19], Weather Systems [21, 22].

Computational simulations can be constructed that approximate these systems and their behaviour according to the governing equations. Exploring the behaviour of these simulations can provide insight into the real-world system they model. This insight can take the form of predicting future states of real-life systems. Such predictions are common in the field of weather forecasting where a model of the weather system is simulated faster than real-time to predict the weather in the following days. Such models are also useful for investigating behaviour of models to identify the optimal conditions for certain phenomena to occur. This can be useful when determining how best to control conditions to achieve a certain system state such as desired structures in alloys, influencing species populations and more.

To computationally simulate the systems governed by these equations, the models must be discretised so they can be stored in computer memory. The continuous fields can be approximated by a lattice of discrete spatial cells where each lattice variable represents the state of the field within that discrete cell. This variable may be a simple numerical type or some complex type depending on the property of the field that must be represented. The equations governing these systems must also be discretised for computation. Methods such as finite-differencing or finite-element can be used to approximate numerical solutions to these equations. These can then be integrated over time to simulate the behaviour of a system according to the model that governs it. The numerical methods used to compute these simulations are discussed in Chapter 3.

The final goal of this research is to determine if numerical simulations of this type of equation can

be constructed automatically from a high-level description of the model. To determine how this can be achieved both the commonalities and differences of these models must be identified. Three equations have been chosen for this purpose - the Cahn-Hilliard equation, the Ginzburg-Landau equation and the Lotka-Volterra equation. Each of these equations has different challenges and peculiarities involved in simulating it. Although these models will not encompass every combination of this type of models, using them as examples does show how simulations of different models can be performed using the same process. Addressing these issues and identifying the properties of each model will lead to a more flexible system that can be used to generate simulations for a wide range of models. These equations are introduced in Sections 2.2, 2.3 and 2.4.

2.2 Cahn-Hilliard Equation

Metallic alloys continue to be an important area of study in materials science. These alloys are a combination of two or more metallic elements [5]. Metal alloys such as steel, aluminium and titanium play a major role in almost every modern engineering project. The properties of these alloys depends highly on the ratio of elements as well as the quenching and aging process. By modeling these alloys during the quenching process, the structures and behaviour can be better understood which can allow insight into how to create alloys with more desirable properties [5].

The Cahn-Hilliard equation provides a useful model of the phase separation of a binary alloy. This equation can be used to explore the structural properties of a quenching binary alloy as it undergoes a phase-transition from an initial 'hot' random state and separates into distinct domains rich in one of the two alloy components. These domains exhibit spinodal decomposition or nucleation depending on the mass fraction of the alloy. These two structures can be seen in Figure 2.1.

The Cahn-Hilliard equation models a binary alloy consisting of A- and B-atoms as a continuous variable u. This scalar field variable represents the excess concentration of A-atoms over B-atoms which ranges between -1 and 1. The extreme value -1 represents an excess of B-atoms while a value of 1 represents excess of A-atoms. The Cahn-Hilliard equation is effectively a mean field approximation of the free energy of a binary system. A full discussion of the Cahn-Hilliard equation can be found in [5,75]. The Cahn-Hilliard equation is given in equation 2.1.

$$\frac{\partial u}{\partial t} = M(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \cdots) \left(-Bu + Uu^3 - K(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \cdots) \right) \tag{2.1}$$

It is convenient to make $M \equiv U \equiv K \equiv 1$ and take B as an inverse temperature parameter in units of the critical temperature. The relationship between B and temperature T is given by:

$$B = \frac{T_c}{T} - 1 \tag{2.2}$$

Any system with a temperature less than the critical temperature T_c (that is when $B > 0$) will separate into distinct A- and B-rich domains. The temperature to which the system is quenched to influences the length scale of the domains and the time they take to form. Initially the system is at an effective infinite temperature and is quenched to the temperature controlled by B. During this quenching process the system will undergo the three stages of quenching [76].

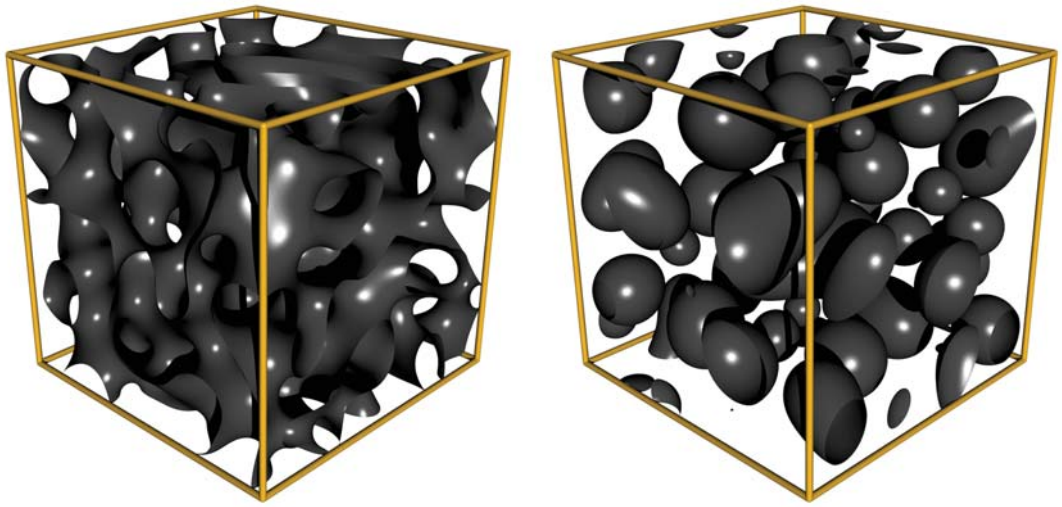


Figure 2.1: Visualisation of two three-dimensional Cahn-Hilliard system showing spinodal decomposition (left) and nucleation (right). These visualisations show the interface between the two A- and B-atom rich domains

Initially the field is in an unstable homogenous state where there are no distinct domains - the first phase. As the field is unstable, it undergoes a phase-transition process where the field separates into distinct A- and B-rich domains - the second phase. After this phase-transition the A- and B-rich domains slowly combine and grow in a process known as coarsening - the third phase. Fields in each of these stages can been seen in Figure: 2.2.

A Cahn-Hilliard system quenched to a colder temperature will move through the first and second stages of quenching faster but will move into the third stage with smaller domains that will take a long time to coarsen. A hotter system closer to the critical temperature T_c will take a longer time to move through the first two stages but will produce larger domains when it enters the coarsening third stage. A system with a temperature of exactly T_c or higher will never form distinct domains as the environment is simply too hot to separate.

The rate at which the domains in a Cahn-Hilliard system form and coarsen depends on the dimensionality of the system. In general, higher-dimensional systems will form domains and coarsen faster. A higher number of dimensions allows the atoms a greater range of motion and allows the domains to coarsen. This dimension-dependent behaviour can be seen in Figure 2.3.

2.3 Time Dependent Ginzburg-Landau Equation

Superconductors are materials that conduct an electric current with no resistance. Superconductivity is a quantum mechanical phenomenon that occurs in certain materials when they are cooled below a certain temperature T_c. Normal metal conductors such as copper can have their electrical resistance

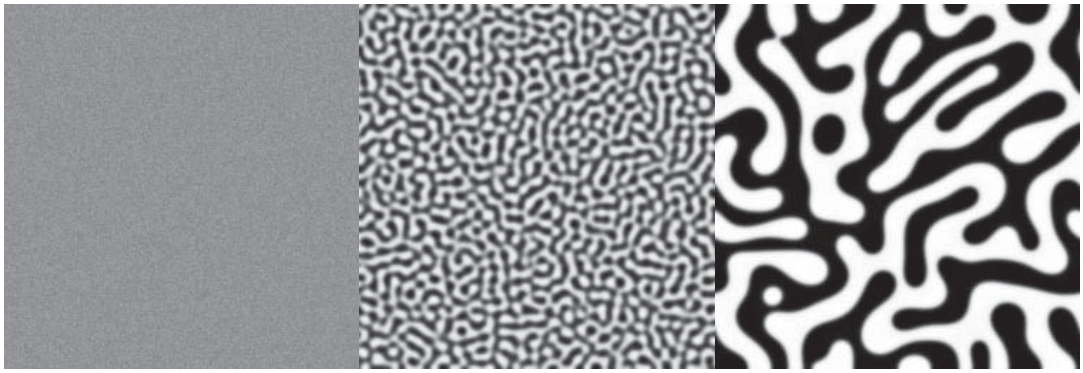


Figure 2.2: From left to right: Phase 1 - initial, unstable homogenous state, Phase 2 - domains start to appear during phase transition, Phase 3 - domains combine and grow during coarsening.

lowered by cooling but can never reach zero resistance due to defects and impurities in the metal. However, superconductors can, in fact, reach a resistance of zero provided they are cooled below T_c.

Ginzburg-Landau theory is a useful framework for describing the macroscopic and thermodynamic properties of superconductors without the need to model microscopic details such as individual atoms, spins, or Cooper pairs [77, 78]. The theory and its uses in describing type II superconductors [79] was developed considerably by Abrikosov [80] and a historical account of the ideas and development was given in the Nobel Lectures of Ginzburg and Abrikosov in 2003 [18, 81]. The Time-Dependent Ginzburg Landau(TDGL) equation and a variation of it known as the Complex Ginzburg Landau equation (CGLE) [82] have been applied and used in a number of applications by various authors since. Ginzburg-Landau theory also arises as the scaling limit for the XY model. The Time-Dependent Ginzburg-Landau equation is given in equation 2.3.

$$\frac{\partial u}{\partial t} = -\frac{p}{i}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\cdots\right) - \frac{q}{i}\left|u\right|^2 u + \gamma u \tag{2.3}$$

In this equation the superconducting field is approximated by a single lattice u where the lattice variables are complex numbers. Each lattice cell value represents how deeply into the superconducting phase the system within that discrete cell is. Numerical simulations of the TDGL can provide useful insight into the behaviour of superconductors. Visualisations of TDGL simulations in two- and three-dimensions can be seen in Figures 2.4 and 2.5.

As u is a complex field variable, it is possible to reformulate the Time-Dependent Ginzburg-Landau equation as two coupled equations that compute the real and imaginary parts of the equation separately. The dynamics of the model itself remain unchanged but storing and computing the model as two coupled scalar fields and equations can have benefits for computational performance [78]. Equation 2.3 can be reformulated as:

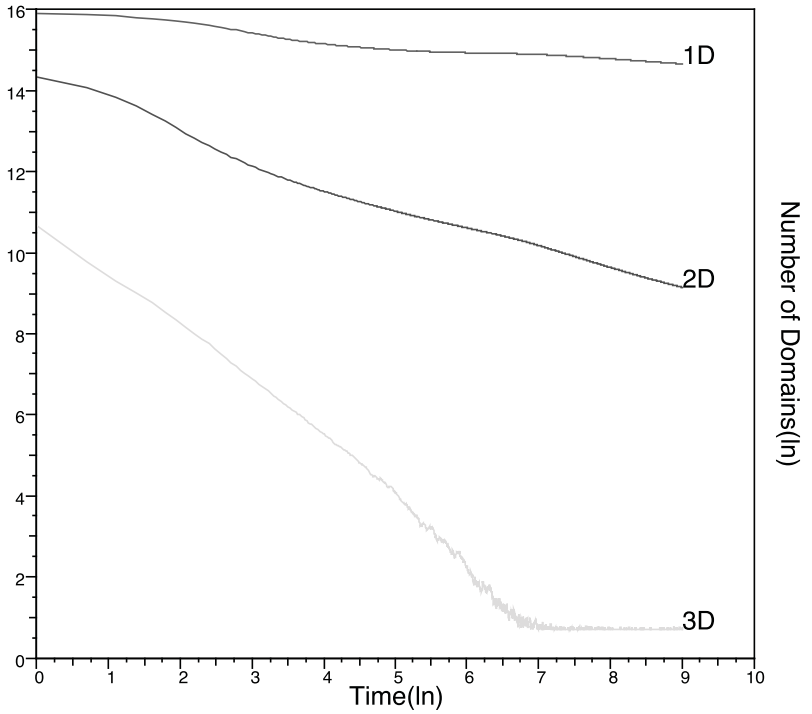


Figure 2.3: Domain coarsening behaviour of the Cahn-Hilliard model. Results shown for 1D, 2D and 3D in ln-ln scale.

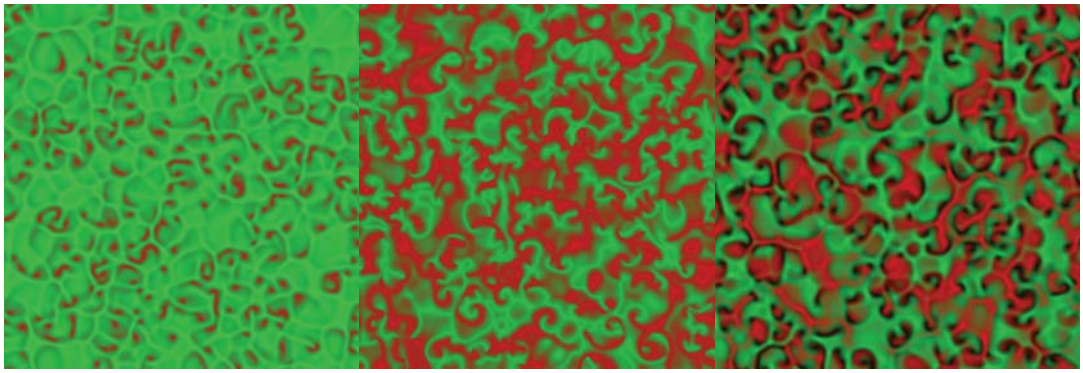


Figure 2.4: Three visualisation methods of the Ginzburg-Landau equation. The visualisations show the absolute value of the field (left), the phase of the field (middle) and the two values combined (right).

$$\begin{aligned}\frac{\partial u_r}{\partial t} &= -p_i\left(\frac{\partial^2 u_r}{\partial x^2}+\frac{\partial^2 u_r}{\partial y^2}\cdots\right)-p_r\left(\frac{\partial^2 u_i}{\partial x^2}+\frac{\partial^2 u_i}{\partial y^2}\cdots\right)\\ &+ -q_i(u_r^3-u_i^3)-q_r(u_r u_i^2+u_r^2 u_i)\\ &+ yu_r\end{aligned} \tag{2.4}$$

$$\begin{aligned}\frac{\partial u_i}{\partial t} &= p_r\left(\frac{\partial^2 u_r}{\partial x^2}+\frac{\partial^2 u_r}{\partial y^2}\cdots\right)-p_i\left(\frac{\partial^2 u_i}{\partial x^2}+\frac{\partial^2 u_i}{\partial y^2}\cdots\right)\\ &+ q_r(u_r^3-u_i^3)-q_i(u_r u_i^2+u_r^2 u_i)\\ &+ yu_i\end{aligned} \tag{2.5}$$

where:
$_r$ is the real component
$_i$ is the imaginary component

This model is useful as an example equation because of its field variable type. Storing and computing complex arithmetic requires the generative programming system to address the issues of arbitrary data types. Using this test case ensures that no unintentional restriction on type has been introduced to the system.

2.4 Lotka-Volterra Equation

The Lotka-Volterra model consists of a set of coupled differential equations that model the periodic cycles of species populations. This set of equations was developed independently by Alfred Lotka [16] and Vito Volterra [17] and is referred to as the Lotka-Volterra model. This model can

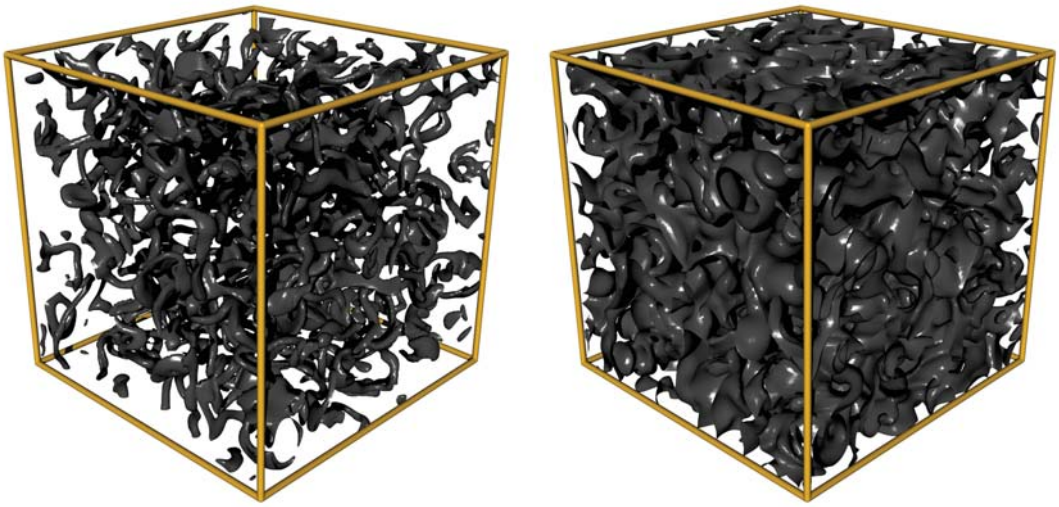


Figure 2.5: Visualisations of a three-dimensional Ginzburg-Landau system showing the absolute value of the field (left) and the phase of the field (right).

be successfully used to model a variety of complex ecological, biological and environmental systems [83].

These equations can be used to model the interaction of predator-prey populations. Such populations are rarely stable and exhibit periodic behaviour. A large prey population provides a large source of food for the predator population. This abundance of food leads to the growth of the predator population which in turn causes the prey population to decrease due to hunting. A decrease in the food source leads to competition within the predator species and the decrease in the population because of lack of food. With this reduced predator population the prey species will start to increase in numbers and the cycle will continue. This population cycle can be seen in Figure 2.7.

The Lotka-Volterra model is not restricted to a two-species system and can be used to describe the interactions of the populations of n different species. The populations in such a system are modeled as an n vector of species populations and the equation written in the form:

$$\frac{d\mathbf{u}}{dt} = \mathcal{F}(\mathbf{u}) \tag{2.6}$$

However, one of the most common formulations of the Lotka-Volterra model is the two-species predator-prey model. Such a predator-prey system can be formulated by expanding equation 2.6 into a system of equations for two species. The governing equations for this system is given in equation 2.7.

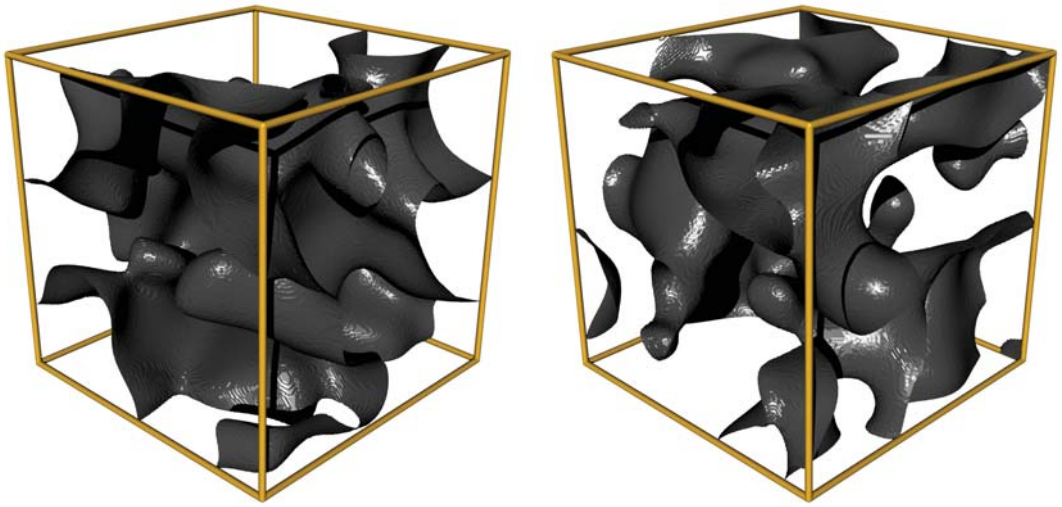


Figure 2.6: Visualisations of a three-dimensional Lotka-Volterra system. These visualisations show the isosurfaces in the populations of the prey (left) and the predators (right).

$$\begin{aligned} \frac{du_0}{dt} &= Au_0 - Bu_0u_1 \\ \frac{du_1}{dt} &= Du_0u_1 - Cu_1 \end{aligned} \tag{2.7}$$

where:
u_0 is the population of prey
u_1 is the population of predators

The parameters A, B, C and D can be used to control the interactions of the species. A controls the growth rate of the prey or u_0. B controls the rate at which the predators 'kill' the prey. C is the rate at which the predators grow from killing the prey. Finally D is the rate at which the predators die.

These systems can be modeled as individual populations but a spatial aspect can also be introduced into the model. This spatial aspect then allows the model to be used to investigate emergent spatial patterns such as spirals [84] and wavefronts [85]. In such a field each cell has its own predator/prey population. To extend this model to approximate a field of predator/prey populations, some form of spatial coupling term between the cells is required. Equation 2.7 can be reformulated to include this spatial coupling term $\mathcal{O}$.

$$\begin{aligned} \frac{du_0}{dt} &= D_0\mathcal{O}(u_0) + Au_0 - Bu_0u_1 \\ \frac{du_1}{dt} &= D_1\mathcal{O}(u_1) + Du_0u_1 - Cu_1 \end{aligned} \tag{2.8}$$

These spatial terms couple neighbouring populations; this term is usually formulated as a spatial average or the gradient of the field [85]. This will transform the equation into a reaction-diffusion system [85]. The parameters D_0 and D_1 are used to control the influence of this spatial term. The formulation of the equation using the gradient of the field is shown in equation 2.9.

$$\begin{aligned} \frac{du_0}{dt} &= D_0(\frac{\partial^2 u_0}{\partial x^2} + \frac{\partial^2 u_0}{\partial y^2}\cdots) + Au_0 - Bu_0u_1 \\ \frac{du_1}{dt} &= D_1(\frac{\partial^2 u_1}{\partial x^2} + \frac{\partial^2 u_1}{\partial y^2}\cdots) + Du_0u_1 - Cu_1 \end{aligned} \tag{2.9}$$

Figure 2.7 shows the behaviour of this formulation of the Lotka-Volterra equation. The plot shows the predator/prey populations of a single lattice site reach a steady cycle around the point $(2, 2)$. This cycle will show variations depending on the interaction with neighbouring populations.

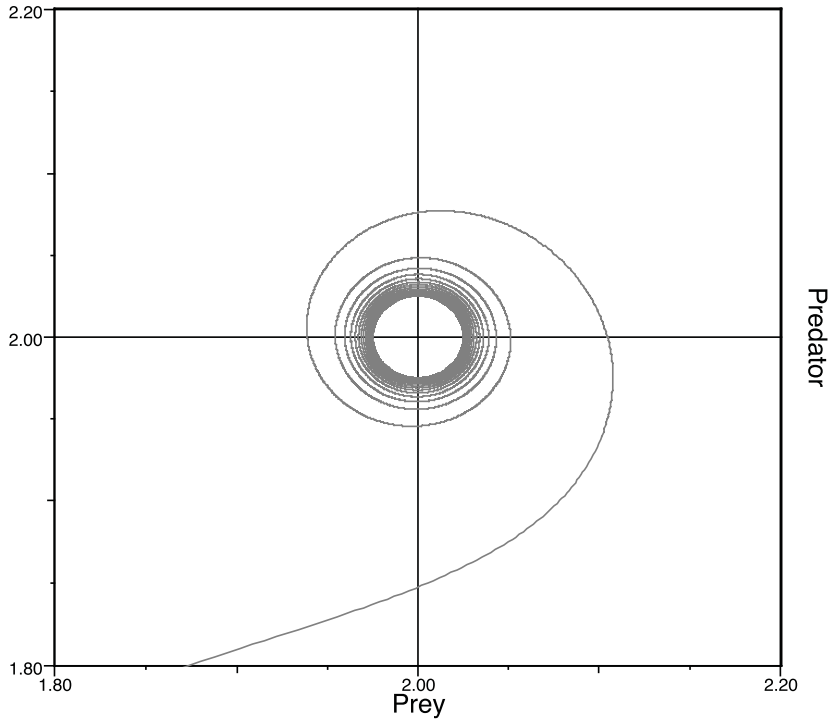


Figure 2.7: The species populations of one site in a Lotka-Volterra system. The plot shows the populations descend into a steady cycle.

The Lotka-Volterra model provides a useful case to test a generative programming system. In this model the field is represented by two lattices and governed by two coupled equations. The system should be capable of generating simulations for models with fields represented by any number of lattices and governed by any number of equations. This model provides a useful test which acts as proof of concept.

2.5 Model Commonalities

This class of partial differential field equations all have a number of commonalities. The state of the system they model can be represented with one or more fields approximated by discrete lattices of various types. These fields are governed by one or more equations, the time derivative of the equations can be defined in terms of the lattice, spatial terms and additional parameters. These commonalities in equation serve as a basis for determining equations which can be used successfully with the system. To identify how to automate the process of constructing simulations, the commonalities of the computation must be determined.

Identifying the commonalities between the simulations of these models, the numerical methods used to approximate solutions to them must be discussed. The same numerical methods can be applied to these three equations and any others of this type. These methods are discussed in detail in Chapter 3.

2.6 Conclusions

This class of partial differential field equations can be used to model a wide range of biological, chemical, physical, mathematical and abstract systems. While each of these models are different and present different challenges for numerical simulations, a number of commonalities between them have been identified. Determining these commonalities is a vital step in automating the construction of numerical simulations of such models.

Three such models have been presented and discussed - the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra models. These three models describe vastly different systems with different properties. Various challenges must be addressed when constructing a numerical simulation of each of these models. Numerical methods and the process of numerically computing approximate solutions to these equations is discussed in Chapter 3.

"All the mathematical sciences are founded on relations between physical laws and laws of numbers, so that the aim of exact science is to reduce the problems of nature to the determination of quantities by operations with numbers."

James C. Maxwell

3 Numerical Methods

3.1 Introduction

Exploring the behaviour of the equations presented in Chapter 2 can provide insight into the real-world systems they approximate. Solutions of these equations can be approximated through the use of numerical methods. The study of numerical methods for approximating solutions to differential equations is a vast topic [62]. This chapter does not attempt to address the entirety of numerical methods but rather focuses on the specific methods used throughout the rest of this thesis. The methods used were selected because of their suitability for parallelisation and automation which is discussed further in Chapters 5, 7, 8 and 9.

To computationally simulate systems governed by these equations, the models must be discretised for storage and computation. The continuous fields of the models can be approximated by a lattice of discrete spatial cells where each lattice variable represents the state of the field within that discrete cell. These lattices can be easily constructed for any dimension where each cell represents a discrete area (2D), volume (3D) etc.

The equations must be also be discretised for computation. Finite-difference methods can be used to approximate the derivatives of these partial differential equations. This allows partial differential equations to be transformed into an ordinary differential equation form that can then be numerically integrated over time. Like the lattices approximating the field, these discrete derivative approximations can be constructed for any dimensionality. This allows these systems to be numerically simulated in one-, two-, three- or higher-dimensions to investigate the dimension-specific behaviour or properties of the models.

Another method commonly used to provide approximate solutions to partial differential equations is the finite-element method. Finite-element methods are more general and applicable to a wider range of equations, especially those using irregular meshes. The models considered in this thesis are the simulation of computational models on regular rectilinear lattices. For these models, finite-differencing provides a suitable and more simple solution.

Many computational models of this type exhibit logarithmic behaviour which require large lattice sizes and/or many simulation time steps to appear. To compute these simulations in a reason-

able time frame, modern parallel computers must be utilised. Finite-difference methods have been used because they are suitable for use with regular rectilinear systems and are significantly easier to parallelise than finite-element methods. For this reason finite-differencing is used to approximate numerical solutions for all simulations.

The ordinary differential equations (ODEs) produced by applying these finite-difference methods cannot be solved analytically. Numerical integration methods must be used to provide an approximate solution. These methods can be split into two categories - implicit and explicit. Both types of methods approximate the state of the system at a later time often by computing intermediate stages. Explicit methods approximate the solution after the time step using only values from the current system state or previously calculated intermediate stages. Implicit methods calculate the state of the system after the time step by solving an equation which involves both the states at the start and end of the time step. These implicit methods must have some method of solving a set of equations using the unknown future state.

Implicit methods are more complex and harder to implement but are still often used because explicit methods can require very small time steps to remain numerically stable. The more expensive implicit methods can be used with larger time steps which results in an overall faster simulation. However, this work uses explicit integration methods because they are much easier to parallelise and do not require iterative solving within the stage computations. Although they require smaller time steps and more function evaluations, they can be executed efficiently on parallel computers and thus are more suitable for this work. In particular the Runge-Kutta integration methods are used as discussed in Section 3.4.

3.2 Finite-Differencing

The three example equations given in Chapter 2 - Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra (shown in equations 2.1, 2.3 and 2.8) are all non-linear in nature, however it is possible to use finite-difference methods to transform them into a form that can be integrated numerically over time. Finite-differencing approximates the spatial terms of these equations to transform them into ordinary differential equations. These ordinary differential equations can then be numerically integrated over time. This allows the dynamics and behaviour of these equations to be simulated.

The spatial terms $(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \cdots)$ in equations 2.1, 2.3 and 2.8 can be replaced with the Laplace operator ∇^2. This Laplace operator is an n-dimensional second order differential operator which is defined as the sum of the second partial derivatives in each dimension. This is given in equation 3.1.

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \cdots \tag{3.1}$$

This Laplace operator has a discrete approximation which can be used to discretise the continuous equations. These discrete approximations can then be used in conjunction with the discrete lattice approximation of the continuous fields. These discrete approximations can then be stored and computed numerically. The discrete form of this operator as applied to a discrete lattice $\mathbf{u}$ is given in

equation 3.2.

$$\nabla^2 \mathbf{u} = \frac{\mathbf{u}_{x,y-h} + \mathbf{u}_{x-h,y} - 4\mathbf{u}_{x,y} + \mathbf{u}_{x+h,y} + \mathbf{u}_{x,y+h}}{h^2} \tag{3.2}$$

It can be shown that this discrete approximation will correspond to the continuous Laplacian operator from equation 3.1 as $h \to 0$. It is often convenient to use a grid size of $h = 1$ which leads to a simple operator which can be easily formulated for any number of dimensions. The form of this operators in one-, two- and three-dimensions as is shown in Figure 3.1.

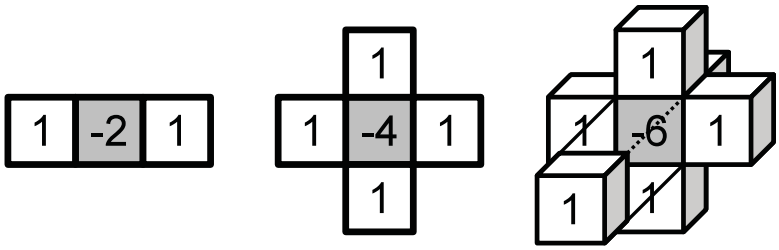


Figure 3.1: The Laplacian operator in one-, two- and three-dimensions.

The discrete forms of the three example equations can be formed by replacing the continuous field variable u with a lattice of discrete cells $\mathbf{u}$ and replacing the spatial derivatives $(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \cdots)$ with the discrete Laplace operator ∇^2. The discrete approximations of the the Ginzburg-Landau and Lotka-Volterra equations are given in equations 3.3 and 3.4.

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{p}{i} \nabla^2 \mathbf{u} - \frac{q}{i} |\mathbf{u}|^2 \mathbf{u} + \gamma \mathbf{u} \tag{3.3}$$

$$\begin{aligned} \frac{d\mathbf{u}_0}{dt} &= \nabla^2 \mathbf{u}_0 + A\mathbf{u}_0 - B\mathbf{u}_0\mathbf{u}_1 \\ \frac{d\mathbf{u}_1}{dt} &= \nabla^2 \mathbf{u}_1 + D\mathbf{u}_0 u_1 - C\mathbf{u}_1 \end{aligned} \tag{3.4}$$

Substituting the discrete Laplace operator is simple for the case of the Ginzburg-Landau and Lotka-Volterra equations; the Cahn-Hilliard equation is somewhat less trivial. The Cahn-Hilliard has two nested Laplace operators as seen in equation 3.5.

$$\frac{\partial \mathbf{u}}{\partial t} = M\nabla^2 \left(-B\mathbf{u} + U\mathbf{u}^3 - K\nabla^2 \mathbf{u}\right) \tag{3.5}$$

This equation can be expanded to the form:

$$\frac{\partial \mathbf{u}}{\partial t} = M \left(-B\nabla^2 \mathbf{u} + U\nabla^2 \mathbf{u}^3 - K\nabla^4 \mathbf{u}\right) \tag{3.6}$$

Which now includes a ∇^4 operator which is known as the biharmonic operator. The operator (in three-dimensions) is shown in its discrete form in Figure 3.2.

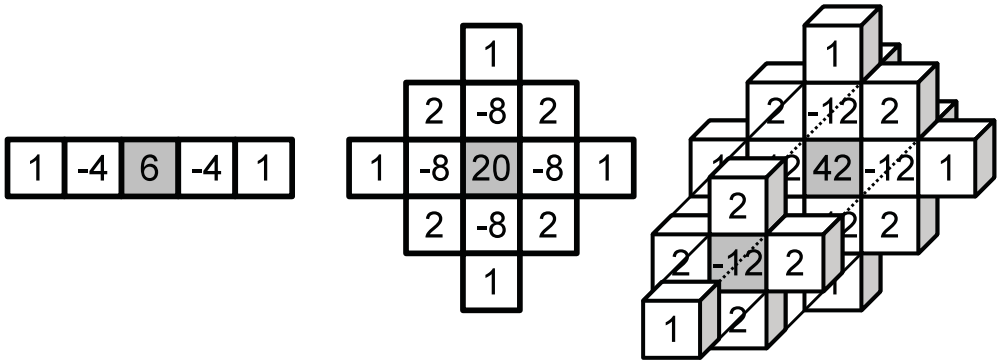


Figure 3.2: The Biharmonic operator in one-, two- and three-dimensions.

These discrete operators and lattices allow the equations to be transformed such that they can be calculated and stored by a computer. Computing these equations gives an approximate derivative of each cell in the lattice. To numerically simulate the behaviour of these equations, the systems must still be integrated in time. There are many possible time-integration methods which can be used. These methods are discussed in depth in Section 3.4.

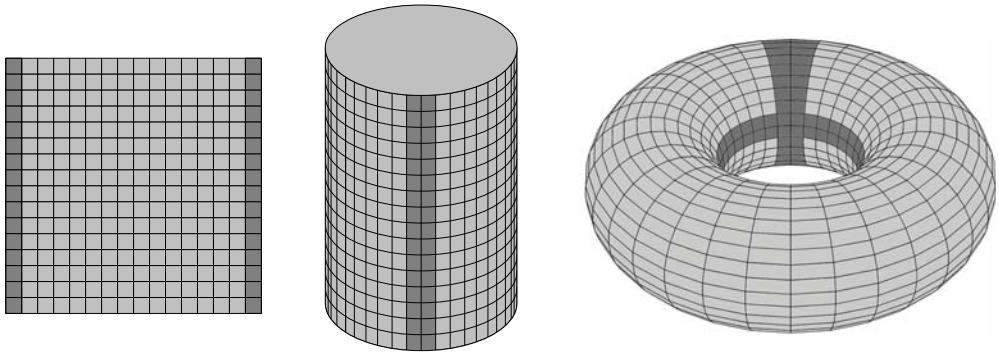
3.3 Boundary Conditions

The differential equation(s) that govern a system are not by themselves sufficient to define a solution. Additional conditions must be imposed to allow a solution to be found. The conditions that can be imposed either separate the problem into a boundary condition problem or an initial value problem. A boundary condition problem specifies information about the system along the boundaries and a solution is searched for that fulfils these conditions. This thesis considers initial value problems which define the initial state of the system and models how it evolves over time [62].

Most of the time systems will be initialised to some random state and then simulated over time to see how the system evolves from this initial state. Because the discrete lattices must be finite, the boundary conditions as well as the initial conditions of these systems must be defined. The solutions found now represent the state of the system at time t that satisfies both the initial and boundary conditions. Three basic boundary conditions are presented.

3.3.1 Periodic Boundary Condition

Periodic boundary conditions are a convenient way of avoiding the problem of defining the behaviour of the system at a boundary. The lattice periodically wraps around on itself or connects to itself in each dimension. This effectively models the system outside the boundaries as an identical copy of the simulated area. The system space is transformed into an infinitely repeating lattice. As this work considers the statistical behaviour of the described models on rectilinear lattices, this method of handling boundaries is often used.



rainin

Figure 3.3: Periodic boundaries applied to a two-dimensional lattice. The dark grey cells 'wrap around' to connect to the corresponding cells on the other border of the lattice. Three images are the original lattice (left), with periodic boundaries in x-dimension (centre) and in both the x- and y-dimensions (right).

Figure 3.3 is an illustration of periodic boundary conditions applied to a two-dimensional lattice in the x-dimension and x- & y-dimension. The dark-grey cells on each border connect to the cells on the opposite border. Periodic boundary conditions effectively allow for the simulation of an infinite lattice. This condition is only suitable for lattices that are sufficiently large that the spatial features of the model will not interfere with each other across the periodic boundaries.

Small lattices can often not sustain spatial behaviour and patterns as the boundary conditions may stop them from forming. If the vortices in the Ginzburg-Landau model span across a lattice with periodic boundaries, the vortex will interact with itself and it cannot be supported. This is one of the situations where large lattice sizes are necessary for the simulation. There are other situations where periodic boundaries are not suitable. For instance a Cahn-Hilliard system with a temperature gradient cannot use periodic boundaries in the dimension of the gradient as it will cause discontinuity in the temperature function. In such a case a different boundary condition must be enforced.

3.3.2 Dirichlet Boundary Condition

The Dirichlet boundary condition is named after Johann Dirichlet [86] enforces a fixed value along the boundary of the system. This type of boundary condition provides a function or set of values to which the cells of lattice on the border must conform. This type of boundary condition is very simple to implement as the value of the boundaries are fixed. For a system u with a range of $[0, 1]$ this type of condition can be expressed as:

$$\begin{aligned} u(0) &= \alpha_1 \\ u(1) &= \alpha_2 \end{aligned} \tag{3.7}$$

where:

α_1 is the value of the system at 0
α_2 is the value of the system at 1

These values must be provided by the boundary condition. This type of boundary condition is easy to implement. The cells of the lattice on the boundary of the system are simply fixed to the values given by the condition.

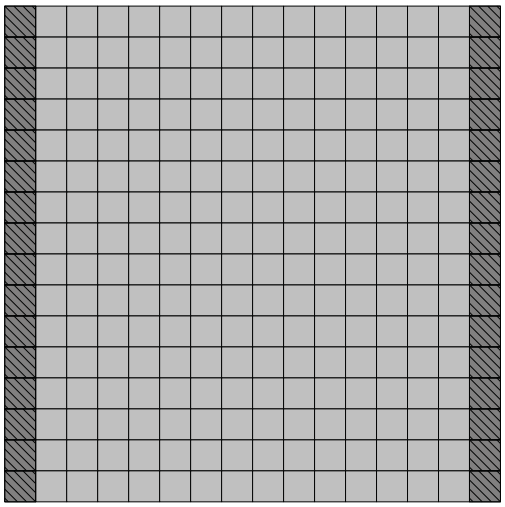


Figure 3.4: Dirichlet boundaries applied to a two-dimensional lattice. The dark grey cells represent border cells that have a fixed value applied to them.

Figure 3.4 is an illustration of a Dirichlet boundary applied in the x-dimension to a two-dimensional lattice. The dark grey cells on each border are set to fixed values. The interpretation of the meaning of enforcing such a boundary will differ depending on the model and what it represents but such a boundary condition is commonly used.

3.3.3 Neumann Boundary Condition

Neumann boundary conditions were named after Carl Neumann [86] and specify the gradient a solution must take at the boundary of the domain. Again, assuming a system u with a range of $[0, 1]$, this type of boundary condition can be defined as:

$$\begin{aligned} \frac{du}{dx}(0) &= \alpha_1 \\ \frac{du}{dx}(1) &= \alpha_2 \end{aligned} \tag{3.8}$$

where:

α_1 is the derivative of the system at 0
α_2 is the derivative of the system at 1

These values must be supplied when defining the boundary condition. Figure 3.5 is an illustration of a two-dimensional lattice with Neumann boundaries applied in the x-dimension.

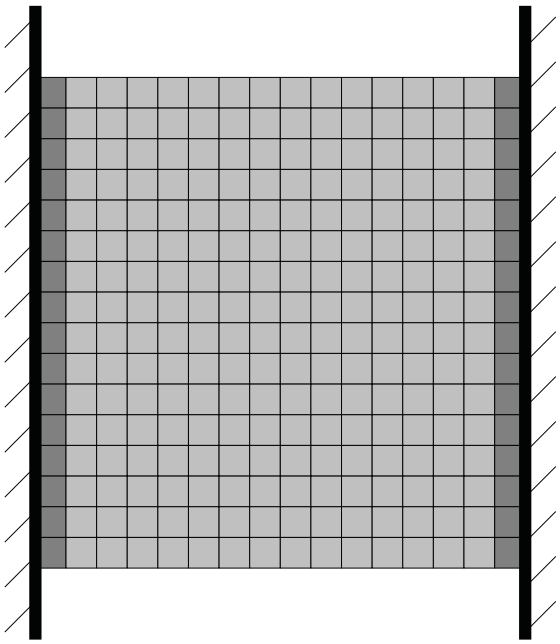


Figure 3.5: Neumann boundary conditions applied in the x-dimension to a two-dimensional lattice. The grey cells are border cells of the lattice where the gradient is set to a fixed value.

This boundary condition can be implemented by changing the stencil applied to the field. The normal operator is given by equation 3.2. If a Neumann boundary is enforced that $\frac{du_0 y}{dt} = \alpha_1$, this will become:

$$\nabla^2 = u_{x,y-1} + u_{x+1,y} + u_{x,y+1} - 3u_{x,y} + \alpha_1 \tag{3.9}$$

It is a relatively simple process to rotate this stencil to apply the Neumann boundaries in other dimensions. Like the Dirichlet the interpreted meaning of applying such a boundary will depend on the model and the value of α_1 but is a useful boundary for many different models.

3.4 Time Integration Methods

Applying spatial finite differencing and boundary conditions allows the models to be discretised for computation. However, these equations must still be integrated over time. There are many different

methods that can be used to integrate these equations. This section focuses on Runge-Kutta methods with both fixed and adaptive stepsizes.

The Runge-Kutta integration methods refer to a whole family of implicit and explicit methods for numerically solving ordinary differential equations. This family of methods was originally developed by the German mathematicians Carl Runge [60] and Martin Kutta [61] but many authors have identified specific methods within this family. These methods provide approximate solutions to ordinary differential equations with various orders of accuracy. Some important higher-order methods include those published by Fehlberg [87], Verner [88] and Dormand and Prince [89, 90]

3.4.1 Euler Method

The most simple numerical integration method that fits within the Runge-Kutta family is the explicit Euler method [62, 91]. This method was described by Leonard Euler in "Institutiones Calculi Integralis" published in 1768. The Euler method approximates the derivate of the equation over the entire time step with the derivative value at the start of that time step. This method is very simple and fast but has very poor accuracy as it is only a first-order method. This integration method can be represented by the following equation:

$$y_{t+h} = y_t + h \times f(y_t) \tag{3.10}$$

where
y_t is the value at time t
h is the time step
$f(y_t)$ is the derivative of y_t at time t

This method is only first order accurate and quickly introduces error into the solution it produces. This error can be reduced by decreasing the time step of the method. However, reducing the time step requires more iterations of the method to compute the solution at the desired time. Because the Euler method is so inaccurate, it is rarely used.

3.4.2 Midpoint Method

A more accurate method is the midpoint method which approximates the derivate over the entire time step as the derivative in the middle of the time step. This method requires two evaluations of $f(y_t)$. The function is evaluated at the beginning of the time step and is used to provide an approximation of the value at the centre of the time step. The derivative of this value is then evaluated and used to compute the final approximation. This method can be represented in the form:

$$y_{t+h} = y_t + h \times f(y_t + \frac{h}{2} f(y_t)) \tag{3.11}$$

The midpoint method is not the only second order accurate Runge-Kutta method. Another second order accurate method in the Runge-Kutta family was developed by Karl Heun [92]. This method is given in equation 3.12.

$$y_{t+h} = y_t + \frac{h}{2} \times (f(y_t) + f(y_t + hf(y_t))) \tag{3.12}$$

3.4.3 Runge-Kutta 4^{th} Order method

A higher order method is the Runge-Kutta 4^{th} Order method which has attained such popularity that it is often referred to simply as the Runge-Kutta method. This method uses three intermediate stages to provide a fourth order accurate solution. This integration method is shown in equation 3.13.

$$\begin{aligned} k_1 &= f(y_t) && (3.13)\\ k_2 &= f(y_t + \frac{h}{2}k_1) && (3.14)\\ k_3 &= f(y_t + \frac{h}{2}k_2) && (3.15)\\ k_4 &= f(y_t + hk_3) && (3.16)\\ y_{t+h} &= y_t + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) && (3.17) \end{aligned}$$

Higher order Runge-Kutta methods can be constructed by computing additional intermediate stages and using them to provide a more accurate approximation of the final value. The different methods in the Runge-Kutta family differ in terms of the number of intermediate stages and the coefficients used to calculate the solution. John Butcher developed a simple and compact way of representing these different methods.

3.5 Butcher Tableaux

All Runge-Kutta methods approximate a solution by combining multiple derivatives calculated from a number of intermediate values. A generalised way of representing them as a set of coefficients can be defined. These methods compute s intermediate stages Y_0, $Y_1 \cdots$, Y_{s-1} and s derivates F_0, $F_1 \cdots$, F_{s-1} where $F_i = f(Y_i)$. Each Y_i is calculated by combining values of F_j added onto the initial value y_t:

$$Y_i = y_t + h \sum_{j=0}^{i-1} a_{ij} F_j \tag{3.18}$$

Once all values of Y_i and F_i have been calculated, the approximation of the final value can be then computed with:

$$y_{t+h} = y_t + h \times \sum_{i=0}^{s-1} b_i F_i \tag{3.19}$$

The distinction between different methods is the value of s and the values of a and b. As such they can be represented as a table of coefficients known as a Butcher tableau. Table 3.1 shows the

general form of these tableaux representing an explicit integration method. Note that all explicit Runge-Kutta tableaux are lower triangular.

Table 3.1: General form of a tableau representing an explicit integration method.

0					
c_1	$a_{1,0}$				
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_{s-1}	$a_{s-1,0}$	$a_{s-1,1}$	$\cdots$	$a_{s-1,s-2}$	
	b_0	b_1	$\cdots$	b_{s-2}	b_{s-1}

These tableaux allow the coefficients for Runge-Kutta integration methods to be easily defined. For example the Euler integration method can be represented by the Butcher Tableau in Table 3.2.

Table 3.2: Butcher Tableau representation of the Euler integration method.

0	
	1

This becomes increasingly useful for representing higher-order Runge-Kutta methods. The Runge-Kutta 4^{th} Order method can be reasonably represented as a set of equations (see equation 3.13, but this becomes increasingly difficult for the even higher-order methods. The RK4 method can be represented quite easily by a Butcher tableau which is shown in Table 3.3.

Table 3.3: Butcher Tableau representation of the Runge-Kutta 4^{th} Order integration method

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

3.6 Commonalities in Numerical Methods

This chapter has described two main numerical methods - finite-differencing and explicit Runge-Kutta integration. From the description of these methods, a number of commonalities can be identified. These commonalities are vital to determining how these methods can be dealt with automatically.

The finite-differencing methods used to approximate the spatial terms in the equations can all be approximated by discrete stencil operators. Solutions to the equations can be approximated by substituting the appropriate discrete stencils into the spatial terms of the equations. Nested spatial terms within the equations must be resolved and can be performed automatically by applying nested stencils to each other.

The calculations to perform an explicit Runge-Kutta integration method can be constructed from a table of coefficients using equation 3.18. All of these explicit methods compute the stages and final state of the system from existing stages according to these calculations. The process of constructing and computing these integration methods is regular and thus can be automated.

3.7 Higher-Order Runge-Kutta Methods

High order integration methods have been described by a number of authors [62, 87–89] and more. There are many possible variations of Runge-Kutta methods with the same order of accuracy yet with different coefficients. For example, the Runge-Kutta 4^{th} order accurate method shown in Table 3.4 as described in John Butcher's book "Numerical Methods for Ordinary Differential Equations" [62] can be used as an alternative to more commonly used method in Table 3.3.

Table 3.4: Butcher Tableau representation of the Runge-Kutta 4^{th} Order integration method

0				
$\frac{1}{4}$	$\frac{1}{4}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	1	-2	2	
	$\frac{1}{6}$	0	$\frac{2}{3}$	$\frac{1}{6}$

These fourth-order accurate methods can be calculated with four function evaluations. Higher order Runge-Kutta methods require increasingly more evaluations to achieve higher orders of accuracy. Two examples are fifth and sixth order accurate integration methods given in Butcher's book [62] which require six and seven function evaluations respectively. The coefficients for these two methods are given in Tables 3.5 and 3.6.

These methods are simply a sample and many even higher order explicit Runge-Kutta methods exist. These higher order methods commonly include an estimation of error to control the stepsize. Further information on these adaptive stepsize methods is presented in Appendix A

Table 3.5: Butcher Tableau representation of a Runge-Kutta 5^{th} Order integration method

0						
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$				
$\frac{1}{2}$	0	0	$\frac{1}{2}$			
$\frac{3}{4}$	$\frac{3}{16}$	$-\frac{3}{8}$	$\frac{3}{8}$	$\frac{9}{16}$		
1	$-\frac{3}{7}$	$\frac{8}{7}$	$\frac{6}{7}$	$-\frac{12}{7}$	$\frac{8}{7}$	
	$\frac{7}{90}$	0	$\frac{16}{45}$	$\frac{12}{90}$	$\frac{32}{90}$	$\frac{7}{90}$

Table 3.6: Butcher Tableau representation of a Runge-Kutta 6^{th} Order integration method

0							
$\frac{1}{3}$	$\frac{1}{3}$						
$\frac{2}{3}$	0	$\frac{2}{3}$					
$\frac{1}{3}$	$\frac{1}{12}$	$\frac{1}{3}$	$-\frac{1}{12}$				
$\frac{5}{6}$	$\frac{25}{48}$	$-\frac{55}{24}$	$\frac{35}{48}$	$\frac{15}{8}$			
$\frac{1}{6}$	$-\frac{3}{20}$	$-\frac{11}{24}$	$-\frac{1}{8}$	$\frac{1}{2}$	$\frac{1}{10}$		
1	$-\frac{261}{260}$	$\frac{33}{13}$	$\frac{43}{156}$	$-\frac{118}{39}$	$\frac{32}{195}$	$\frac{80}{39}$	
	$\frac{13}{200}$	0	$\frac{11}{40}$	$\frac{11}{40}$	$\frac{4}{25}$	$\frac{4}{25}$	$\frac{13}{200}$

3.8 Conclusions

This chapter has introduced a set of numerical tools and methods for approximating numerical solutions to the partial differential equations described in Chapter 2. The important numerical methods used in this thesis are the discrete finite-differencing stencils which can be substituted into the equations to replace the spatial terms and the time-integration methods used to integrate the equations over time.

A number of commonalities in these methods have been identified. Regular methods for representing them and applying them to the partial differential equations have been found. These commonalities allow the process of constructing numerical simulations from equations and methods to be automated. This automatic process is further discussed in Chapter 7, 8 and 9.

"The only real valuable thing is intuition."

Albert Einstein

4

Parallel Architectures and Languages

4.1 Introduction

Computing the numerical simulations of the partial differential field equations can be computationally expensive. Many spatial patterns and emergent properties of the models are only sustainable by large simulation lattice sizes. Investigating the dimension dependent behaviour of the models can compound this large system size requirement. In order to compute these simulations and produce results in a meaningful time-frame, the power of parallel computers must be exploited.

For four decades Moore's Law [93] has been an accurate predictor of CPU processing speed. However, in more recent years hardware development has experienced a "slowing down" of this law [94–97]. While Moore's Law is still holding true in terms of the number of transistors, chip design is approaching physical limits and the increase of clock speeds has dramatically stalled. CPU producers have been increasing the cache sizes in an attempt to improve performance but the days of simply waiting for a faster core to make a problem computationally feasible have ended [94, 96, 98].

Chip manufacturers have started placing multiple cores on the same die to increase the overall computational power of CPUs [97]. These multiple cores share the same system memory and normally have individual and shared levels of cache. At the time of writing, most modern CPUs contain between two and six cores although some research chips are being developed which contain up to eighty cores [99, 100]. The main challenge faced by chip developers at this point is not how to add more cores, but rather how the CPU architecture can support and utilise these cores [96, 101].

As cores are added to the chip design, managing data becomes increasingly complicated. Data must be kept consistent across several levels of cache which are shared or distributed across the separate cores. Maintaining consistent data across several caches is known as *cache coherency* or *cache consistency*. As cores are added, there are more separate caches and more cores accessing and modifying data. Keeping the caches consistent becomes increasingly difficult as the number of cores is increased, impacting negatively on performance [98].

A rather different approach to incorporating many cores into a single processing unit is the GPU. Initially developed to render the detailed real-time three-dimensional graphics required by the com-

puter games industry, GPUs have adopted an approach of using many low-power cores with little or no memory cache [102,103]. While the computational power of each core is relatively low and access to memory slow (due to the lack of caching), GPUs provide overall high computational throughput due to the sheer number of cores that can be used with this architecture. At the time of writing, modern high-end GPUs contain up to 512 cores. GPU architectures are discussed in detail in Section 4.4.

Another approach to parallelism considers not only the number of cores on a single machine but a number of machines connected by a network. These machines are known as *distributed computers* as the memory of the machine is distributed across the different nodes on the network. Distributed machines can be constructed with various network structures for many different applications [104–107]. Cluster, Grid and Cloud computers are all based on this general distributed architecture.

The major advantage of such machines is their scalability. Large network structures can be created to allow more machines to collaborate on tasks. Clouds and Grids often have nodes distributed across different cities, countries and continents and often use the internet to communicate. The nodes of modern distributed computers are often parallel machines in their own right [108]. These networked machines may contain multi-core processors, multiple processors or parallel accelerator devices such as GPUs. At the time of writing the most powerful computer on the Top500 is the Tianhe-1A which is built with nodes containing six-core CPUs and GPU accelerators [109].

Regardless of the specific parallel architecture and language used, the parallelisation of the simulations is vital. Once again it is important to identify the common elements of how these simulations can be parallelised to determine how parallel implementations of the simulations can be generated for various parallel architectures. To identify how this can be accomplished, some common parallel architectures and the languages used to program them are introduced.

4.2 Central Processing Units

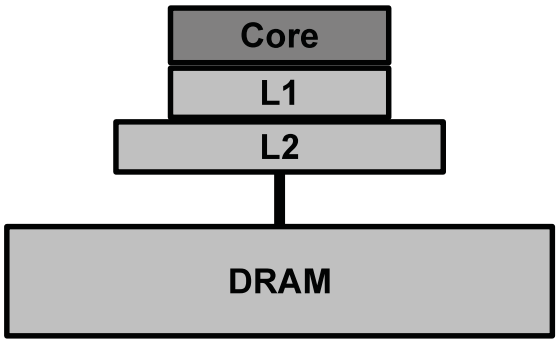


Figure 4.1: Single-Core CPU architecture, against which many applications programmers optimise their software.

For many years CPUs have contained a single processing core with cached access to the system memory. This common design is summarised in Figure 4.1. This type of CPU design is easy to write code for as there is only a single core accessing, changing and writing values and one cache

hierarchy. Multi-core CPUs will generally have some form of individual and shared memory cache all connected to the shared system memory [96, 98]. Figure 4.2 shows an example design of a quad-core CPU. Each CPU core has its own L1 cache with L2 cache shared between all four cores.

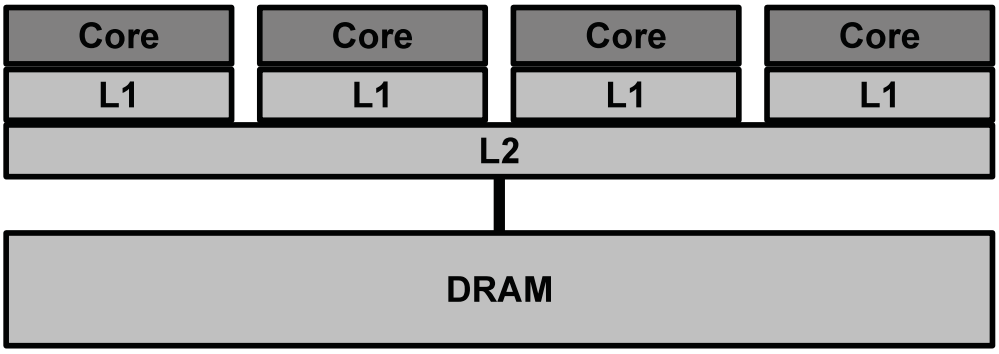


Figure 4.2: Multi-Core CPU architecture, four cores with their own L1 caches and a shared L2 cache.

One of the major challenges which must be overcome by multi-core chip designers is cache coherency between cores [98, 108]. Modern CPUs rely on several levels of cache to keep the cores supplied with the necessary data. The different cores of a CPU have some of these cache levels shared between them and some are specific to that core. When multiple cores are manipulating the same data, these different levels of cache must be kept coherent. This becomes increasing difficult as the number of cores in the CPU is increased. More cores means more caches that must be kept coherent and this will have a negative impact on overall performance of the CPU.

With CPU design adopting multi-core architectures on a large scale [94, 96, 101] multi-threaded programming is becoming increasingly important. Developers can no longer rely on having faster cores to allow for more complex applications and instead must design their programs to make use of multiple-cores. Parallel programming will have to move away from high-level courses learnt by experts and become a more commonly-used technique [94].

There are many languages and libraries available that allow multi-threaded code to be written in a variety of different ways. In this thesis two C-based libraries are discussed - POSIX threads [110] and Threading Building Blocks [111].

4.2.1 POSIX threads

POSIX Threads or Pthreads is a low-level library for using system threads [110]. This library allows for the explicit creation of threads and inter-thread communication. This low-level method of programming threads can often provide the best performance, however it can require a great deal of parallel knowledge and is also the most error-prone.

When implementing a program using Pthreads, the main considerations are ensuring that the threads co-operate together to perform the computation correctly. As threads must often perform computation using the same data, they must be careful not to interfere with each other. With the Pthreads library, this coordination must be explicitly programmed by the developer. The two main

constructs used for this are `mutexes` and `semaphores`.

Mutexes allow threads to 'lock' data. When data is accessed by multiple threads, a mutex can be used to stop any other threads accessing and modifying that data out of order. Threads can lock a mutex, modify the data and then unlock the mutex again. If a thread tries to lock a mutex while it is already locked by another thread, it will wait until the other thread has unlocked the mutex before it can continue. Through the correct use of mutexes, it can be guaranteed that only one thread will be accessing that piece of data at any time.

Semaphores allow threads to communicate and synchronise with each other. When a thread waits on a semaphore, it will halt until another thread posts to that semaphore to continue. These semaphores have a counter counting the number of posts and waits. A post will increment this counter while a wait decrements it. If the semaphore counter is positive, a wait will continue immediately and decrement the counter. If it is 0, the thread will halt until the semaphore receives a post.

These low-level methods of Pthreads are very fast and closely tied to the functionality of the CPU. Unfortunately the correct use of them is hard to learn and race-conditions can easily be introduced into a program. Another approach to multi-threaded programming is to use a higher-level library such as Threading Building Blocks.

4.2.2 Threading Building Blocks

Intel's Thread Building Blocks (TBB) provides a high-level way of writing a multi-threaded application. Rather than managing threads explicitly (although this is possible) TBB provides a library of commonly-used parallel constructs and functions [111]. This still requires the developer to have knowledge of parallel programming to identify how the program can be safely parallelised but makes the implementation far less complex and error-prone.

TBB provides parallel algorithms that can be invoked without the need for manually managing thread synchronisation and data access. One such algorithm is `parallel_for` which will iterate in parallel over some range (specified at run-time) and perform some computation as defined by the user. Other algorithms such as `parallel_sort` will use the available cores on the CPU to sort a list of items. This style of multi-threaded programming is more limited but is much easier to program and learn.

TBB also provides more low-level constructs such as `mutexes`, `task_groups` and `threads` [111]. Thus TBB can be used for more complex low-level problems but the programmer must address the issues of data access and synchronisation manually, as with the Pthreads library.

TBB contains many different functions and constructs which allows a wide range of parallel programs to be implemented with it. The full functionality of TBB is presented in [111].

4.3 Cluster Computers

Cluster computers have been a very popular and successful supercomputer design for many years and are useful for a wide range of applications [104, 106, 106, 108]. Rather than building a single

powerful machine, cluster computers attain computational throughput by connecting many simple machines together with a network interconnect. One of the major advantages of cluster computers is the scalability of the design. More nodes can be easily added to the machine by extending the network with additional switches. One simple cluster configuration can be seen in Figure 4.3.

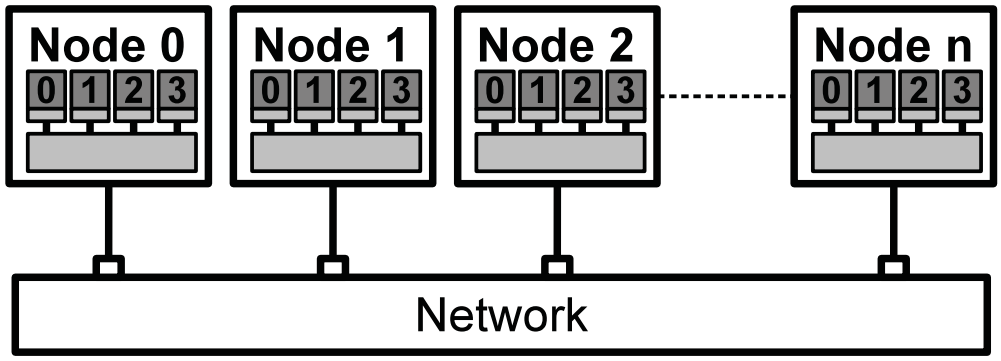


Figure 4.3: A cluster computer architecture showing quad-core nodes connected together by a single switch.

There are many possible ways to configure a cluster machine, from the configuration of individual nodes to the design of the interconnect network [107]. The example cluster shown in Figure 4.3 contains nodes which each contain a quad-core CPU. Generally each core of a multi-core CPU is treated as a separate node which allows applications to be developed the same way regardless of the actual cluster configuration on which they are executed.

The other major design consideration is the network structure [107]. Clusters are normally connected together by switches - as the number of nodes increases it is not possible for all the machines to be connected to a single switch. There many different possible network designs such as hierarchical structures, hypercubes, tori and many more which can each provide different benefits and drawbacks [104, 105, 107].

The main challenge of writing applications for cluster machines is how to decompose the memory storage and computation across many nodes and manage the communication between them. The communication across a network is relatively slow which can often lead to a high communication/-computation ratio. This is especially true as the number of nodes increases and the network structure connecting the nodes together grows more complex. Minimising this ratio is vital for achieving good performance for an application executing on a cluster.

4.3.1 Message Passing Interface

For cluster machines to collaborate together on a single computational task, they must communicate via the network. The Message Passing Interface or MPI is an API specification that has been developed for this purpose [112,113]. MPI defines a set of methods that allow nodes to identify themselves, synchronise with each other and exchange data with each other through the sending and receiving of messages.

There are various implementations of MPI: MPICH [114], LAM [115], OpenMPI [116] and others. These implementations differ in terms of optimisations for the size and frequency of messages sent between nodes. Different implementations often provide additional methods which are not part of the MPI standard. These proprietary features differ between implementations but all should provide the base functionality defined by the MPI standard.

4.4 Graphical Processing Units

In recent years, GPUs have emerged as a very powerful, parallel computing architecture. Graphics card development has been driven and funded by the computer games industry. The cards were originally designed purely for rendering the increasingly detailed and complex three-dimensional graphics of real-time computer games. GPU manufacturers long ago adopted a parallel architecture to meet these demands.

Although these GPUs were initially very focused towards computing the rendering pipeline, they have slowly moved towards more general processing architectures to allow for more advanced rendering methods to be incorporated into computer games. In recent years, a number of APIs have started to emerge to allow these GPUs to be used for non-graphics applications [102]. This is known as General Purpose computation on GPUs or GPGPU. The development of GPGPU APIs such as NVIDIA's CUDA [103], BrookGPU [32], ATI's Stream SDK [117] and sh [33] have made GPGPU applications increasingly easy to develop. These libraries allow developers to implement applications in C-style syntax that can then be executed on GPU devices.

GPUs can be very effective at accelerating a wide range of scientific applications [102, 118]. The performance benefits they offer depends on the application's suitability for fine-grained parallel decomposition but in many cases can provide as much as a 50-100x speed-up over a conventional CPU core. Much of the research into computer science over the last several decades has been focused on developing methods for parallelising applications, thus there exist many well-known methods for decomposing a variety of applications into parallel tasks.

GPUs have data-parallel architectures reminiscent of parallel computers from the 1980's such as the Distributed Array Processor [26, 119] and the Connection Machine [120]. As most of the parallel computing research over the last decade has been focused on developing applications for distributed architectures, a certain degree of adaptation is required to transfer those parallel designs to the data-parallel architecture of GPUs. Most of the research into GPU programming involves finding the optimal way to solve a problem on the data-parallel architecture while making the best use of the optimisations specific to the GPU architectures.

One of the most inviting features of the GPU as a parallel computing architecture is the ease of accessibility. GPUs are relatively cheap and provide easy access to a highly-parallel and powerful processing architecture. This coupled with the easy to use GPGPU APIs allows a large user-base access to high-power parallel applications with little start-up cost. Their computing power to cost ratio is also highly desirable - a $500 (US) graphics card has the potential to provide 50-100x speed-up over a single-core of a CPU [102].

4.4.1 NVIDIA's CUDA

With the rise of General Purpose computation on Graphical Processing Units (GPGPU), a number of APIs have been developed - CUDA [103], BrookGPU [32], Stream [117] and sh [33]. Out of these APIs, CUDA stands out as the most developed, powerful and widely used. CUDA has been developed by NVIDIA and thus is only compatible with NVIDIA graphics cards. CUDA and NVIDIA graphics cards are used throughout this thesis to implement GPU applications.

During the course of this research there have been two major architecture versions of CUDA compatible GPUs - the Tesla and Fermi architectures. Tesla was the first GPU architecture capable of executing CUDA applications and the newer Fermi architecture GPUs are a similar but improved design with additional features. Each of these architectures is discussed in the following two sections.

4.4.2 NVIDIA Tesla Architecture

The Tesla architecture is the base design for a range of NVIDIA cards that encompass the GeForce 8 series, 9 series and 200 series graphics cards as well as NVIDIA's professional Quadro series and Tesla computing cards. Tesla architecture GPUs contain a scalable array of multi-threaded processing units known as Streaming Multiprocessors (SMs) [31]. Gamer-level GeForce cards can contain up to 30 of these SMs while simple desktop cards typically contain between 2-4. Each Tesla multiprocessor contains 8 Scalar Processors (SPs) which are used to execute the actual instructions. Multiprocessors are capable of creating, managing and executing threads in hardware resulting in almost no scheduling overhead [31].

Because Tesla architecture GPUs do not contain the high levels of cache used by modern CPUs to reduce memory latency, memory is the limiting performance factor for many CUDA applications. To improve performance of such applications, multiprocessors also contain several types of memory that are optimised for different purposes. These memory types allow threads executing on the same multiprocessor to share data. The correct use of this on-chip memory can have a large impact on performance. These memory types are as follows:

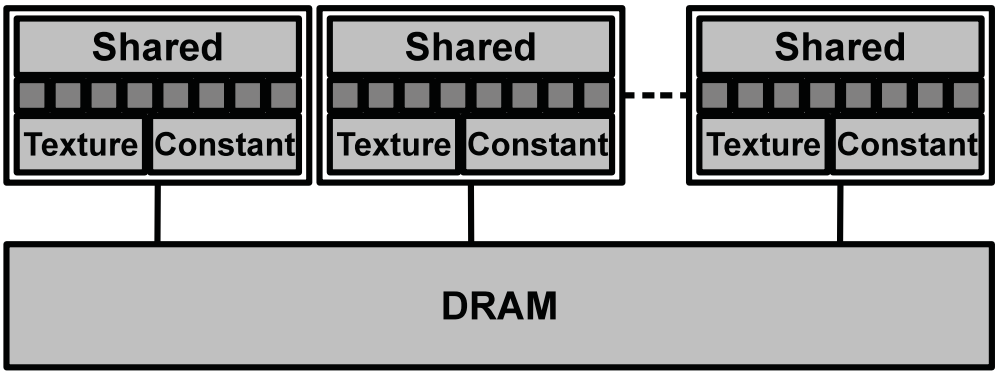


Figure 4.4: A Tesla architecture GPU with two Streaming Multiprocessors each containing eight Scalar Processors. This architecture can be easily extended to contain many more multi-processors.

- **Global Memory** - the largest section of memory stored on the graphics card and the only memory type that the CPU can read from and write to. This memory is accessible by all threads but also has the slowest access times (400-600 clock cycles). The overall time taken to fetch data from global memory can be improved by using a technique called coalescing. When 16 sequential threads access 16 sequential and word-aligned memory addresses, the transactions can be coalesced into one single read/write to Global memory.

- **Registers** - on-chip memory used to store the local variables belonging to each thread. Provided there are at least 192 active threads, the access time for registers is effectively zero extra clock cycles [31]. Registers are used automatically and the only consideration for the programmer is how many registers are used by each thread. If the registers cannot store all the local variables required by the threads, Local Memory must be used. Devices of compute capability 1.0 and 1.1 have 8192 registers per multiprocessor while compute capability 1.2 and 1.3 have 16384 [31].

- **Local Memory** - actually stored in Global memory, automatically used to store any local variables that cannot fit into the registers. Access times are the same as accessing Global memory; all transactions will be coalesced.

- **Shared Memory** - read/write memory stored on-chip that can be used to share data between threads executing on the same multiprocessor. Provided that the threads access different memory banks, the memory transactions are as fast as reading from a register. Each multiprocessor has 16KB of shared memory organised into 16 memory banks [31].

- **Texture Memory** - on-chip cached method of accessing Global memory. Global memory bound to a texture will only be read if the desired value is not already stored in the texture cache. If so the cache is reloaded with that value and the values surrounding it in the spatial locality defined by the texture dimensions. On a cache hit, the transaction is as fast as reading from registers. Textures can be cached in one-, two- or three-dimensions. Texture cache size is between 6-8KB per multiprocessor [31].

- **Constant Memory** - another on-chip cache for accessing Global memory, a Global memory read is only required in the event of a cache miss. This memory type is designed to allow all threads to read the same value at the same time. If all threads read the same value from constant memory, the cost is the same as reading from registers. Each multiprocessor has 8KB of constant cache [31].

As these different types of memory must be used explicitly by the developer, a great deal more consideration of memory access is required when developing a GPU application as compared to a traditional CPU implementation. Correct use of these memory types is vital to the performance of a GPGPU application [102]. Correct use is not limited to the type of memory used but also the access patterns used by the threads.

4.4.3 NVIDIA Fermi Architecture

The Fermi GPU has been specifically designed by NVIDIA for GPGPU. The advances of Fermi over the previous architecture GPUs include: improved performance for double precision calculations, more shared memory, more SPs per multiprocessor and cached Global memory access [103, 121]. Each multiprocessor on a Fermi architecture GPU contains 32 Stream Processors and 8x the peak processing power for double precision floating point calculations over the previous generation. Along with an L1/L2 cache structure this architecture represents a significant step forward in GPGPU. This cache structure can be seen in Figure 4.5.

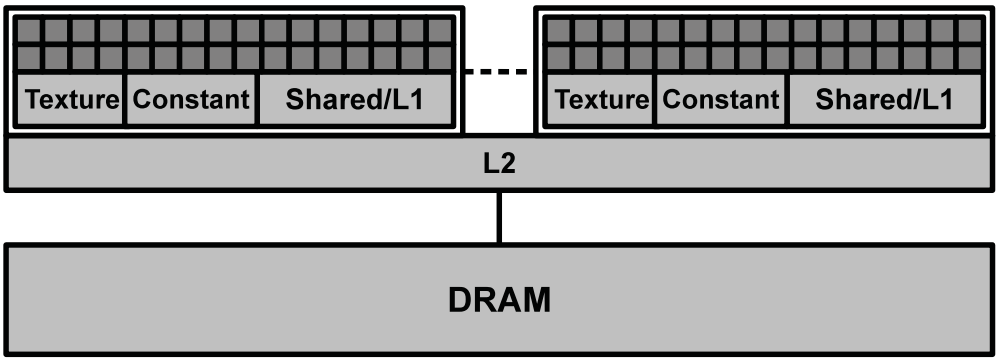


Figure 4.5: A Fermi architecture GPU with showing multiprocessors containing 32 SPs and the new cache structure.

This cache structure makes CUDA applications development more accessible to the average developer as it can improve memory access speeds for global memory. In previous architectures, the explicit use of texture/constant cache and shared memory were necessary to provide good performance [102]. The Fermi tuning guide recommends the use of cached global memory access rather than using the texture cache due to the higher bandwidth of the L1 cache [122].

Fermi multiprocessors have 64KB of on-chip memory which is used by shared memory and the L1 cache. This memory can be configured to 48KB for shared memory and 16 KB of L1 cache (the default setting) or vice versa. This allows the developer to dedicate more memory to the L1 cache if less shared memory is required.

There are several differences in the behaviour and size of the device memory between the Fermi and Tesla architecture GPUs. These differences are as follows:

- **Global Memory** - coalesced memory access is achieved when 32 threads access sequential and aligned memory addresses rather than 16. Access to global memory is also cached in the L1/L2 cache structure.
- **Registers** - all Fermi GPUs of compute capability 2.0 contain 32768 registers rather than 8192 or 16384.
- **Local Memory** - stored in global memory in case all registers are full. All access to local memory will be cached in the L1/L2 cache.

- **Shared Memory** - stored in the same memory as the L1 cache. Can be configured to prefer either L1 or shared memory. Depending on configuration, shared memory size is either 16KB or 48KB. Fermi shared memory has 32 memory banks rather than 16.

- **L1/L2** - automatic cache hierarchy for accessing global and local memory. L1 cache is stored in the same location as shared memory and can be configured to either 16KB or 48 KB. The L2 cache is 768 KB and services all load, store and texture requests.

This improved memory performance along with the increased overall core count, memory speed and clock speed make Fermi architecture GPUs far more powerful GPGPU devices. GPGPU applications for both the Tesla and Fermi architectures can be written using NVIDIA's CUDA.

4.4.4 CUDA applications

In order to parallelise an application effectively for computation on a GPU, it must be decomposed into tasks (kernels) that can be executed in parallel. Rather than splitting the computation evenly between each core as is common with most parallel architectures, the computation is split in as fine-grained a way as possible. In general the number of threads created will match the number of data elements being processed rather than the number of cores. The advantage of creating many more threads than there are cores is that the GPU can schedule thread executions to help hide memory latencies.

The programmer has no direct control over how these threads are distributed to the cores on the GPU. However the programmer can control how threads are grouped together into blocks which will then be executed together on the same multiprocessor. In this way the programmer can control which threads will execute together and share data through Shared memory and the Texture and Constant caches. Each block has three dimensions (x,y,z) and can contain no more than 2^9 threads for the Tesla architecture and 2^{10} for Fermi. All of these blocks are organised into a two-dimensional grid which has a maximum size of 2^{16} in each dimension.

When an application is launched the thread blocks will be distributed among the multiprocessors. When a block is assigned to a multiprocessor, it will split them into groups of 32 threads known as *warps*. Every time an instruction is issued, the multiprocessor will select a waiting warp and issue it an instruction. This programming model is known as SIMT or Single-Instruction Multiple-Thread [31] as the SPs on a single multiprocessor all execute the same instruction at the same time but different multiprocessors are able to execute instructions independently.

By controlling the size and shape of the blocks, the programmer can indirectly control how memory will be accessed. For example, the width of a thread block should generally be a multiple of 16 (32 for Fermi). This way each set of 16 threads will access sequential memory addresses (assuming each thread accesses one value from a global array) and thus global memory transactions can be coalesced (32 threads for the Fermi architecture).

One situation that can have a serious impact on performance is branches. If the threads in a warp branch to different instruction paths, the paths will be executed serially by disabling threads not on that path until the paths converge. For this reason it is beneficial to ensure that all the threads that

will be part of the same warp follow the same branch path whenever possible. Failure to do so will have a significant impact on the performance of the application.

4.5 Multi-GPU

While GPUs are a powerful parallel architecture, they are limited in terms of total computational throughput and memory [56]. A single high-end GeForce card can provide a theoretical peak performance of 1.58 Tflops and 1536MB of global memory - Tesla computing cards can provide 1.03 Tflops with 6GB of global memory. To overcome these limitations, multiple GPU devices must be used. To use multiple GPUs for a single task, the computation must be split between the devices. For most problems, some communication and synchronisation between the devices will be required. For many multi-GPU systems there is no way to directly communicate between GPUs; all communication must be performed through the CPU [56]. However, direct device-device access is possible on multi-GPU systems that are equipped with compute capability 2.0 Fermi GPUs and use CUDA 4.0 [123].

Many motherboards contain multiple PCI-E x16 slots which allow them to host multiple graphics cards. Also some graphics cards contain two GPUs (such as the GeForce GTX295 or GTX590). In both these cases, separate CPU threads are required to connect to each GPU device. Figure 4.6 shows one possible configuration of four single GPU cards connecting to a quad-core CPU. PCI extender chassis such as the Dell PowerEdge C410x [124] can increase the number of GPUs to which a single host can connect. The PowerEdge C410x allows a single host to connect to 16 graphics cards [124].

Any communication and synchronisation between GPUs must be managed by the controlling CPU threads using an appropriate multi-threading library. To communicate information between GPUs, their controlling host threads must first copy the data out of the GPU, then exchange it with another CPU thread which will copy it into its GPU. Alternatively if the system is capable of direct device-to-device communication, the GPUs can either access the values in the other device memory or the CPU threads can copy data directly between GPUs. Using either system, multi-GPU programming requires knowledge of both GPU and multi-threading programming.

Individual threads must be created to connect to each GPU device and manage the data communication and synchronisation between the devices. GPU devices can only connect to a single host thread at a time and for this reason Multi-GPU is better suited to direct multi-threading libraries such as Pthreads rather than task-based libraries like TBB. Many of the TBB constructs are not applicable and the programmer must overcome the challenges of multi-threaded programming manually - for this reason all multi-GPU applications in this thesis use CUDA and Pthreads.

These systems encounter the usual problem of communication versus computation time. For multiple GPUs to be used efficiently, the time taken to perform the computation should be larger than the time required to communicate the necessary data between them. For multi-GPU systems, this requires very large problem sizes as each GPU is in itself a powerful, highly-parallel computer. The best method for improving this communication to computation ratio is obviously dependent on the problem; however, CUDA does provide a very useful piece of functionality that can be useful for many applications.

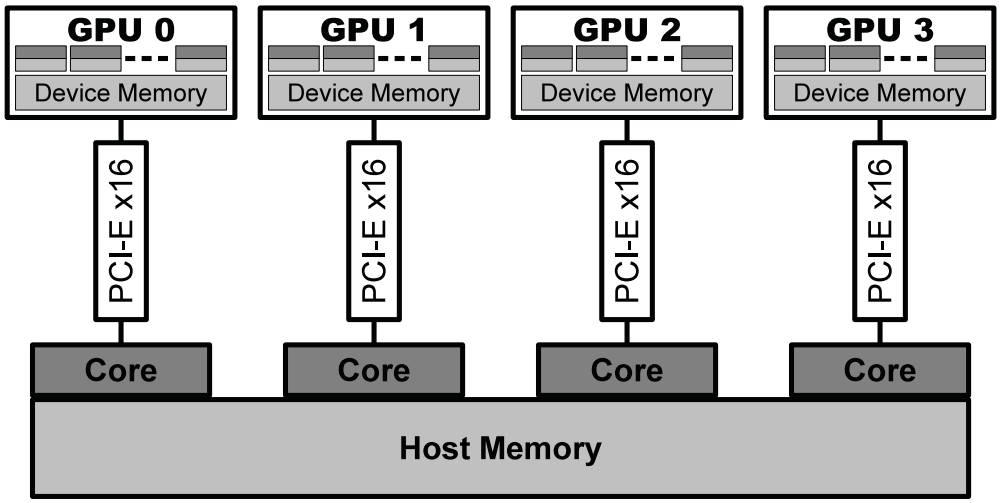


Figure 4.6: A multi-GPU machine containing four GPUs each hosted on separate PCI-E x16 buses and managed by separate CPU cores.

CUDA has support for asynchronous execution and data-transfer [31,103]. This type of execution allows the GPU threads to execute instructions at the same time as the CPU transfers data in and out of GPU memory. Using this functionality can allow the communication to be performed at the same time as the computation which, in certain conditions, can significantly reduce the time spent waiting for communication. Using this functionality has been shown to make the use of multiple GPU devices scale effectively [125–127].

Such Multi-GPU machines have been shown to produce good scalability for a variety of problems such as saliency map models [128], RNA folding algorithms [129] and Smith-Waterman sequence alignment [130]. The scalability of Multi-GPU machines is limited by the number of GPUs that can be hosted on a single machine. Some motherboards are capable of hosting up to four graphics cards which could potentially contain two GPU cores, limiting the maximum number of GPUs on a single machine to eight. As previously mentioned some PCI extender chassis can be used to host up to 16 graphics cards [124].

Such machines would require large power supplies but are still worth considering. How well an application can be divided across GPU cores is, as always, dependent on the application. If no communication between the GPUs is necessary, then a near linear speedup can be expected, however as the communication between the GPUs increases this speedup is expected to decay. In these situations optimal use of functions such as asynchronous memory copy and execution is vital to attaining good scalability across the multiple GPU cores.

4.6 GPU-Accelerated Clusters

While multi-GPU machines do extend the memory and computational limits of GPU devices, they are currently limited by the number of PCI-E x16 slots available on the motherboard or PCI exten-

der chassis. This limitation can be overcome by connecting a number of GPU-accelerated machines together into a cluster. This solution has no limit on scalability as it is always possible to add more nodes. However, as more nodes are added, the time for communication can be expected to increase. This approach has already proved a powerful supercomputer architecture with the top place on the November 2010 TOP500 list going to the Tianhe-1A machine which uses this configuration [109]. A diagram of one possible GPU cluster configuration is shown in Figure 4.7.

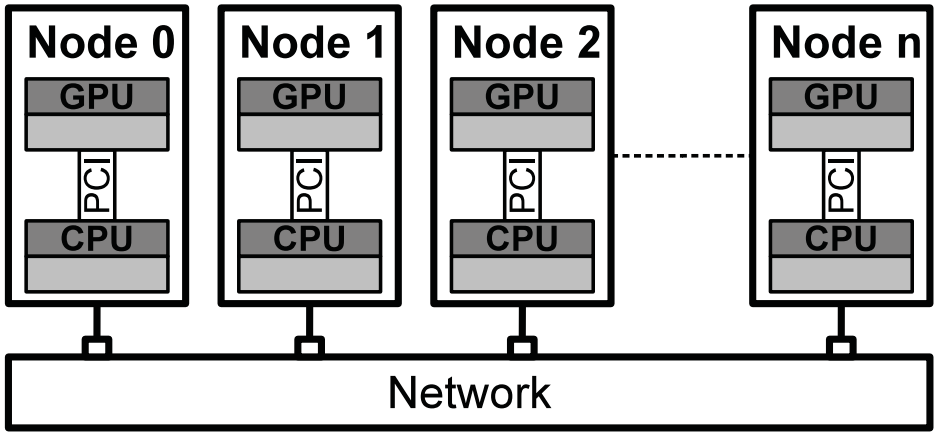


Figure 4.7: A diagram showing a GPU cluster configuration containing n nodes each hosting a single GPU.

GPU clusters require the use of two technologies - the relatively new GPGPU APIs and the well-developed distributed computing APIs such as MPI [112]. GPU clusters with promising scaling results have been described for a number of applications - Finite Differencing [56, 127], Biomedical Analysis [131], lattice Boltzmann model [132], Fluid Dynamics [133].

In these systems, the communication is not only between devices but also between the cluster nodes. The CPU cores in these cluster nodes must perform communication across the connecting network using a communication interface such as MPI [112, 116]. A GPGPU library such as CUDA [31, 103] is used to perform actual computation on the GPU. The nodes on these clusters are not limited to a single GPU and can consist of a distributed architecture of multi-GPU nodes. In such cases there is communication between the GPUs in the same node via the host memory and inter-node communication across the cluster communication interconnect.

Such GPU-accelerated clusters face more challenges in terms of achieving a high computation-to-communication ratio. While the GPUs can be utilised to accelerate the nodes of the cluster to reduce computation time, they can provide no benefit in terms of improving the inter-node communication speeds. This issue has been documented by GPU cluster developers [134]. The GPUs do not reduce the performance of the cluster and are almost always successful in accelerating it [127, 132–135]; however, the limitations of the node communication infrastructure becomes more apparent.

Like multi-GPU machines, GPU-accelerated clusters can make use of CUDA's asynchronous memory copy and execution functionality to perform communication while the GPU continues to perform calculations. This can be used effectively with MPI's asynchronous Send/Recv to commu-

nicate data between GPUs on the same node and between nodes in the cluster without the GPUs being left idle [127, 135].

4.7 Commonalities in Parallel Architectures

While each of these parallel computing architectures is different in nature and each has individual challenges when programming applications for it, they are conceptually similar. All of the parallel architectures contain multiple cores which read data, execute instructions and write to memory. The architectures can be separated into two categories - shared- and distributed-memory machines.

Shared-memory machines store all the data in a single shared memory location which the cores read from and write to. The threads executing on these cores must synchronise with each other to ensure correct computation. Distributed memory machines have separate memory locations which may be accessed by only one or more cores. Threads running on these machines must synchronise with each other and exchange data through some communication channel.

These parallel architectures require the use of a parallel programming language and library. Multiple languages/libraries may be required for hybrid parallel machines. These languages are all different but once again have a number of similarities. To determine how these similarities can be exploited to automatically construct finite-difference simulations using these languages, the specific implementations of the simulations must be reviewed. These implementations are discussed and compared in Chapter 5.

4.8 Conclusions

A number of popular parallel computing architectures have been introduced and described. Some of the languages and libraries used for implementing parallel programs on these architectures have been presented. Each of these architectures and languages has individual strengths and weaknesses in terms of cost, performance, program complexity and scalability.

To utilise the computational processing power of these architectures for simulating computational models, the simulations must be decomposed into tasks which can be processed in parallel. The exact manner of this parallelisation depends on the simulation, the architecture and the language/libraries used to implement them. Algorithms for computing these simulations in parallel are presented in Chapter 5.

The architectures and languages presented in this chapter are only a sample of the current technology available. It can be expected that new languages and architectures will emerge in the future. Thus if code for these simulations is to be generated, the system must be capable of producing code for any language or architecture but also be easily extensible to the new languages which will be developed in the future.

"An algorithm must be seen to be believed."

Donald Knuth

5 Parallel Algorithms

5.1 Introduction

The numerical methods discussed in Chapter 3 describe how the PDEs in Chapter 2 can be numerically discretised in time and space. For these simulations to be computed, the numerical equations must be implemented in programming code. This code must deal with creating, storing and managing the lattices representing the system, computing the equations and integrating the system over time.

The code to compute the equations can be easily written using mathematical operators and code to access the lattice values required by the finite-differencing stencils. Many different integration methods can be used to integrate these systems as discussed in Chapter 3. Each of these methods have different requirements in terms of computational cost and memory usage. Multi-stage methods may require intermediate lattices to be stored and additional stages computation. Algorithms that compute simulations using these multi-stage methods must consider the order of computation and data synchronisation to ensure correct results.

Computing simulations with large lattice sizes and many time steps required to show certain behaviour can be a very computationally-intensive task. The correct use of parallel architectures and languages such as those discussed in Chapter 4 is vital to producing simulation results in a reasonable time-frame. These simulation implementations require the use of various languages and methods of splitting the computation.

To automate the process of constructing these simulation implementations, the common features and the language specific features must be identified. All the different implementations must split the computation into tasks which can be computed in parallel. Identifying which parts of the simulation computation can be parallelised will be common to all the simulations and each implementation will require one of the decomposition methods discussed in Section 5.5. The implementations will also require language and architecture specific supporting code which will not be common to other implementations. Identifying these features is vital to determining how to automatically generate simulations.

There has been a significant amount of research into computing such lattice-based computa-

tions on parallel computers [23]. This work includes implementations on Distributed Computers [21, 57–59, 136, 137], Multicore Processors [138], Graphical Processing Units [53, 56, 139] and GPU clusters [127, 135] with various methods of parallel decomposition [140, 141]. This chapter discusses both sequential and parallel implementations of finite-differencing simulations. Many of these implementations have been discussed in previous publications [75, 78, 102, 125–127, 142] but are collated here.

5.2 Equation Computation

To create any sequential or parallel simulation, the first necessary functionality is to compute the equations for a given system. The numerical methods discussed in Chapter 3 transform the equations into a discrete form which can be stored and calculated by computer. All of the architectures and languages presented in this thesis use C-style syntax. These languages were chosen to make it easy to compare different implementations on different architectures and because of the author's personal preference. One of the advantages of this approach is that evaluating the equations for a system can be performed using almost the same code. The code fragments that evaluate the three equations from Chapter 2 are presented in the following sections.

5.2.1 Cahn-Hilliard Equation

In order to compute the Cahn-Hilliard equation, the neighbouring values required by the discrete stencils must first be fetched from memory. The Cahn-Hilliard equation contains a biharmonic stencil which requires not just the nearest neighbours to be fetched from memory but also their nearest neighbours. These values are then substituted into the equation with the appropriate coefficients from the stencil to compute the equation for a particular cell. Listing 5.1 shows the code that computes the Cahn-Hilliard equation for a single cell (x, y) in a two-dimensional lattice with dimensions (X, Y). The code has been formatted to make the stencil structure easily visible.

Listing 5.1: The C-syntax code to evaluate the Cahn-Hilliard equation at position (x, y) for a discrete two-dimensional floating-point lattice **u** with dimensions (X, Y).

```
float u_ym2x    = u[(y-2)*X +  x   ];
float u_ym1xm1 = u[(y-1)*X + (x-1)];
float u_ym1x    = u[(y-1)*X +  x   ];
float u_ym1xp1 = u[(y-1)*X + (x+1)];
float u_yxm2    = u[ y   *X + (x-2)];
float u_yxm1    = u[ y   *X + (x-1)];
float u_yx      = u[ y   *X +  x   ];
float u_yxp1    = u[ y   *X + (x+1)];
float u_yxp2    = u[ y   *X + (x+2)];
float u_yp1xm1 = u[(y+1)*X + (x-1)];
float u_yp1x    = u[(y+1)*X +  x   ];
float u_yp1xp1 = u[(y+1)*X + (x+1)];
float u_yp2x    = u[(y+2)*X +  x   ];

M*(
(-B*(      u_ym1x  +
u_yxm1 + (-4*u_yx) + u_yxp1 +
           u_yp1x
)) +
(U*(                    (u_ym1x*u_ym1x*u_ym1x) +
(u_yxm1*u_yxm1*u_yxm1) + ( -4*u_yx*u_yx*u_yx  ) + (u_yxp1*u_yxp1*u_yxp1) +
```

```
                          (u_yp1x*u_yp1x*u_yp1x)
)) -
(K*(                       u_ym2x    +
       (2*u_ym1xm1) + (-8*u_ym1x) + (2*u_ym1xp1) +
u_yxm2 + (-8*u_yxm1) + ( 20*u_yx ) + (-8*u_yxp1) + u_yxp2 +
       (2*u_yp1xm1) + (-8*u_yp1x) + (2*u_yp1xp1) +
                          u_yp2x
)));
```

It should be noted at this point that this code does not account for boundary conditions. Executing this code without extra handling for boundary conditions could cause array index out of bounds errors. Implementing boundary conditions in code is discussed in Section 5.3.

5.2.2 Ginzburg-Landau Equation

The Time-Dependent Ginzburg-Landau equation can be computed in a similar fashion. The TDGL equation uses the smaller Laplacian stencil which requires only the nearest-neighbour values to be fetched from memory. It should be noted that **u** is a lattice of complex numbers and this code assumes a complex number class where appropriate operators have been provided. The code to compute the TDGL equation can be seen in Listing 5.2.

Listing 5.2: The C-syntax code to evaluate the Time-Dependent Ginzburg-Landau equation at position (x, y) for a discrete two-dimensional complex lattice **u**.

```
complex u_ym1x    = u[(y-1)*X +  x    ];
complex u_yxm1    = u[ y   *X + (x-1)];
complex u_yx      = u[ y   *X +  x    ];
complex u_yxp1    = u[ y   *X + (x+1)];
complex u_yp1x    = u[(y+1)*X +  x    ];

 - (P/i)*(  u_ym1x  +
 u_yxm1 + (-4*u_yx) + u_yxp1 +
            u_yp1x
)
- (q/i)*
      (abs(u_yx*u_yx)*u_yx)
+            y*u_yx;
```

If there is no complex number class available, the real and imaginary parts of the model must be computed as separate equations. Such a system must be stored as two separate lattices (**u_r** and **u_i**) representing the real and imaginary parts of the field. Performing the computation in this way can also offer performance benefits on some architectures [78]. The code to compute the model in this separated form is given in Listing 5.3.

Listing 5.3: The C-syntax code to evaluate the Ginzburg-Landau equation at position (x, y) using two separate lattices and calculations for the real and imaginary parts. The system is stored in two floating point lattices **u_r** and **u_i**.

```
float uym1x_r   = u_r[(y-1)*X +  x    ];
float uyxm1_r   = u_r[ y   *X + (x-1)];
float uyx_r     = u_r[ y   *X +  x    ];
float uyxp1_r   = u_r[ y   *X + (x+1)];
float uyp1x_r   = u_r[(y+1)*X +  x    ];

float uym1x_i   = u_i[(y-1)*X +  x    ];
float uyxm1_i   = u_i[ y   *X + (x-1)];
float uyx_i     = u_i[ y   *X +  x    ];
```

```
float uyxp1_i   = u_i[ y   *X + (x+1)];
float uyp1x_i   = u_i[(y+1)*X +  x   ];

- p_i*(      uym1x_r  +
 uyxm1_r + (-4*uyx_r) + uyxp1_r +
             uyp1x_r)
- p_r*(     uym1x_i  +
 uyxm1_i + (-4*uyx_i) + uyxp1_i +
             uyp1x_i)
- q_i*(uyx_r*uyx_r*uyx_r + uyx_i*uyx_i*uyx_r)
- q_r*(uyx_r*uyx_r*uyx_i + uyx_i*uyx_i*uyx_i)
+ (y*uyx_r);

  p_r*(      uym1x_r  +
 uyxm1_r + (-4*uyx_r) + uyxp1_r +
             uyp1x_r)
- p_i*(      uym1x_i  +
 uyxm1_i + (-4*uyx_i) + uyxp1_i +
             uyp1x_i)
+ q_r*(uyx_r*uyx_r*uyx_r + uyx_i*uyx_i*uyx_r)
- q_i*(uyx_r*uyx_r*uyx_i + uyx_i*uyx_i*uyx_i)
+ (y*uyx_i);
```

5.2.3 Lotka-Volterra Equation

Finally the code to compute the Lotka-Volterra equation is given in Listing 5.4. This code uses the Laplace operator but now there are two lattices **u0** and **u1** which approximate the system. To compute the equation for the cell at position (x,y), the neighbouring values from both lattices must be fetched. Two separate computations are required to update the cell in both of these lattices. This is similar to the TDGL implementation which stores and computes the real and imaginary parts of the system separately.

Listing 5.4: The C-syntax code to evaluate the Lotka-Volterra equation at position (x, y) for two coupled, discrete two-dimensional floating-point lattices **u0** and **u1**.

```
float u0_ym1x   = u0[(y-1)*X +  x   ];
float u0_yxm1   = u0[ y   *X + (x-1)];
float u0_yx     = u0[ y   *X +  x   ];
float u0_yxp1   = u0[ y   *X + (x+1)];
float u0_yp1x   = u0[(y+1)*X +  x   ];

float u1_ym1x   = u1[(y-1)*X +  x   ];
float u1_yxm1   = u1[ y   *X + (x-1)];
float u1_yx     = u1[ y   *X +  x   ];
float u1_yxp1   = u1[ y   *X + (x+1)];
float u1_yp1x   = u1[(y+1)*X +  x   ];

(A*u0_yx)
-
(B*u0_yx*u1_zyx)
+
(D0*(       u0_ym1x  +
u0_yxm1 + (-4*u0_yx) + u0_yxp1 +
            u0_yp1x
));

(C*u0_yx*u1_yx)
-
(D*u1_yx)
+
(D1*(       u1_ym1x  +
u1_yxm1 + (-4*u1_yx) + u1_yxp1 +
            u1_yp1x
));
```

5.3 Boundary Conditions

Methods for implementing the three simple boundary conditions discussed in Chapter 3 are presented here. These are relatively simple boundary conditions but they are sufficient for the purpose of these example simulations. There are various possible ways to implement these conditions but only one implementation for each is presented here. The following sections discuss the three boundary conditions - Periodic, Dirichlet and Neumann.

5.3.1 Periodic Boundaries

Periodic boundary conditions are very easy to implement in code. All the required neighbouring values are stored in the lattice, only the calculation of their indexes must be changed. If the neighbouring index is less than 0 or greater than the size of the lattice, the boundary condition must be applied. This can be performed by adding or subtracting the lattice length to the computed index. In the x-dimension, the neighbouring index -1 becomes $X - 1$ and X becomes $X - X$.

Listing 5.5 shows a code fragment to apply periodic boundaries to the index calculation in two-dimensions. This code computes the neighbours of a site (x,y) by calculating the two neighbouring indexes of the site in each dimension ($xm1$, $xp1$, $ym1$ and $yp1$).

Listing 5.5: The C-syntax code to apply periodic boundary conditions in two-dimensions.

```
ym1 = (y == 0) ? Y-1 : y-1;
xm1 = (x == 0) ? X-1 : x-1;
xp1 = (x == X-1) ? 0 : x+1;
yp1 = (x == Y-1) ? 0 : y+1;
```

The code in Listing 5.5 uses the ternary operator because of its higher performance compared to `if` statements. This performance different is particularly noticeable for GPU implementations.

5.3.2 Dirichlet Boundaries

Dirichlet boundary conditions can also be applied to a simulation relatively easily. The value of the field outside its boundary is a fixed value or function. If an index is outside the range of the field, the lattice will not be accessed but instead a function is called which returns the boundary value at that point. This may be a fixed value or some computed value.

Listing 5.6 gives an example of how Dirichlet boundaries can be implemented. The code applies Dirichlet boundaries in two-dimensions and uses four functions ($by0(y,x)$, $byY(y,x)$, $bx0(y,x)$ and $bxX(y,x)$) to compute the values on each boundary.

Listing 5.6: The C-syntax code to apply Dirichlet boundaries in two-dimensions.

```
u_ym1x = (ym1 < 0)   ? by0(ym1,x) : u[ym1*X + x];
u_yxm1 = (xm1 < 0)   ? bx0(y,xm1) : u[y*X + xm1];
u_yxp1 = (xp1 >= X) ? bxX(y,xp1) : u[y*X + xp1];
u_yp1x = (yp1 >= Y) ? byY(yp1,x) : u[yp1*X + x];
```

This implementation also uses the ternary operator to either calculate a boundary value or fetch a value from the lattice. Like the periodic boundary implementation, this option was selected for performance reasons.

5.3.3 Neumann Boundaries

Implementing Neumann boundaries are more complex. The exact implementation will depend on the nature of the computation model and the stencils it uses for computation. In general the equation can be reformulated to exclude the lattice site outside the boundary and include the Neumann α term. The exact reformulation will depend on the model but the boundary conditions can be implemented by a series of `if` statements.

Listing 5.7 shows the series of `if` statement for necessary to enforce Neumann boundary conditions on a two-dimensional lattice. There are eight possible conditions when the boundaries are necessary. Four for the boundary in each dimension and another four when boundaries in both directions are encountered.

Listing 5.7: The C-syntax code to enforce Neumann boundary conditions on a two-dimensional lattice.

```
if((y == 0) && (x == 0)) {
        //Top Left
} else if((y == 0) && (x == X-1)) {
        //Top Right
} else if((y == Y-1) && (x == 0)) {
        //Bottom Left
} else if((y == Y-1) && (x == X-1)) {
        //Bottom Right
} else if(x == 0) {
        //Left
} else if(x == X-1) {
        //Right
} else if(y == 0) {
        //Top
} else if(y == Y-1) {
        //Bottom
} else {
        //Interior
}
```

5.4 Sequential Implementation

Programming these simulations on a single-threaded CPU is relatively simple. For every step the CPU will iterate over the lattice or lattices and compute the equation for each lattice cell. Using an appropriate integration method it will compute a new value to be written into the cell of another lattice representing the system after the time step. This iteration process may have to be performed several times during each step depending on the number of stages of the integration method. Implementations of three integration methods described in Chapter 3 are presented for the CPU - Euler, RK2 and RK4.

5.4.1 Euler

The Euler integration method is the least-accurate explicit method, but it is fast, has low memory requirements and is easy to implement. Euler only requires one computation stage per time step. For every time step the CPU will iterate over every cell in the lattice $u0$ and calculate the equation for that cell $f(u0)$. The new value is then calculated using the Euler method ($y_{t+h} = y_t + h \times f(y_t)$) and written to the output lattice $u1$. This will perform a single simulation time step. The code to perform this computation can be seen in Listing 5.8.

Listing 5.8: Code for a finite-differencing simulation using the Euler integration method implemented for a single-core using C.

```
void euler(double *u0, double *u1) {
   for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u1 = u0 + f(u0)*h
      }
   }
}
int main(int argc, char** argv) {
   double u0 = new double[Y*X];
   double u1 = new double[Y*X];
   ...
   for(int t = 0; t < no_steps; t++) {
      cahn_hilliard_euler(u0,u1);
      swap(&u0, &u1);
   }
   ...
}
```

As the Euler integration method only has one stage, only one iteration over the lattice is required. More complex integration methods with multiple stages will require multiple iterations over the lattice to compute the methods. These methods also require additional memory to store the intermediate stages while the Euler method only requires two - the input and output lattices. A simple example of a multi-stage Runge-Kutta integration method is the RK2 method.

5.4.2 Runge-Kutta 2^{nd} Order

The Runge-Kutta 2^{nd} order integration method implementation requires an additional iteration over the lattice and additional memory to store the intermediate stage. Because this method uses the derivative at the midpoint between time steps, the lattice must first be computed. Then the derivative of this midpoint lattice is used for the calculation of the final lattice at the next time step. The implementation of the update function can be seen in Listing 5.9.

Listing 5.9: The single-threaded CPU implementation of the RK2 integration method.

```
void runge_kutta_2nd(double *u0, double*u1, double *u2) {
   for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u1 = u0 + f(u0)*h/2
      }
   }
   for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u2 = u0 + f(u1)*h
      }
```

```
    }
}
```

This simulation implementation will take longer to execute and require more memory than the Euler implementation. Because the midpoint field must be first calculated, two iterations over the lattice are required, so this method can be expected to take approximately twice as long. It will also require an additional lattice to be stored in memory, requiring an additional 50% of memory usage over the Euler method.

5.4.3 Runge-Kutta 4^{th} Order

The Runge-Kutta 4^{th} order method can be implemented in a very similar way to the RK2 method, but the number of stages is increased. The RK4 method has four stages and requires additional lattice iterations and memory space. Because the function evaluations of the RK4 method are reused in the final computation, it is possible to optimise the implementation for either memory usage or speed.

Either the function evaluations can be stored in memory or they can be recomputed during the final computation. Storing the function evaluation of a lattice usually requires the same amount of memory as storing a system lattice. This uses significantly more memory but reduces the computation considerably. There are various tricks which can be used to reduce memory storage while also storing the function evaluations which can be used for the RK4 and high-order Runge-Kutta methods. For ease of presentation the RK4 method which recomputes the function evaluations is shown here in Listing 5.10.

Listing 5.10: The update function of the RK4 integration method implemented in C for a single-core of a CPU.

```
void runge_kutta_4th(double *u0, double*u1, double *u2, double *u3, double *u4) {
   for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u1 = u0 + f(u0)*h/2
      }
   }
   for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u2 = u0 + f(u1)*h/2
      }
   }
    for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u3 = u0 + f(u2)*h
      }
   }
    for(int iy = 0; y < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         // u4 = u0 + f(u0)*h/6 +  f(u1)*h/3 +  f(u2)*h/3 +  f(u3)*h/6
      }
   }
}
```

These different integration methods can be implemented using any of the parallel languages and architectures discussed in Chapter 4. The following sections discuss some of the possible ways to implement these simulations in parallel. These example implementations use the RK2 integration method as it shows the requirements for a multi-stage method and allows the examples to be short and concise.

5.5 Parallel Decomposition

To compute a finite-differencing update in parallel, the computation must be split between threads which execute on different cores of a parallel computer. To do this, the lattice is split into sections and each section is updated by different threads. There are many ways in which a regular rectilinear lattice can be split for distribution to multiple cores [140, 141]. The lattice can be split into the same number of sections as there are cores, or it can be split into more sections and each core will process more than one section. It is also possible to split the field into less sections than there are cores but this obviously leaves some cores with no computation to process. There are also a number of different symmetries that can be used to split the field.

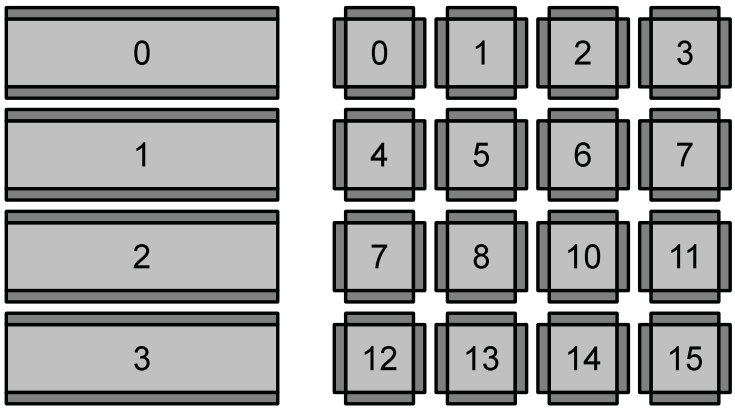


Figure 5.1: Two methods of two-dimensional rectilinear lattice decomposition. The lattice is evenly divided in one-dimension (left) and in two-dimensions (right). The dark grey cells represent the neighbouring cells of the lattice to which each section must have access.

All of the decomposition methods shown here split the lattice into rectilinear sections of equal size. To do this the lattice is divided evenly in one or more dimensions. This gives two methods of decomposing two-dimensional lattices and three methods for three-dimensional lattices. These methods of decomposition are shown in Figures 5.1 and 5.2.

There are a number of ways these symmetries can be rotated to provide other possible permutations of splitting the lattice. Each decomposition method has various benefits in terms of border area, contiguous memory storage et cetera. They provide different benefits for different parallel architectures and implementations. The parallel decomposition methods used by each parallel implementation are discussed in more detail in the following sections.

Because finite-differencing computations require access to the values of neighbouring cells, cores processing neighbouring lattice sections must exchange bordering information after each stage of computation. Different lattice decomposition methods will have different requirements for this communication. The section borders for each decomposition method are represented by the dark grey cells in Figures 5.1 and 5.2. The performance implications of each method will differ in terms of border area, number of neighbours to communicate with and how well the method scales to more

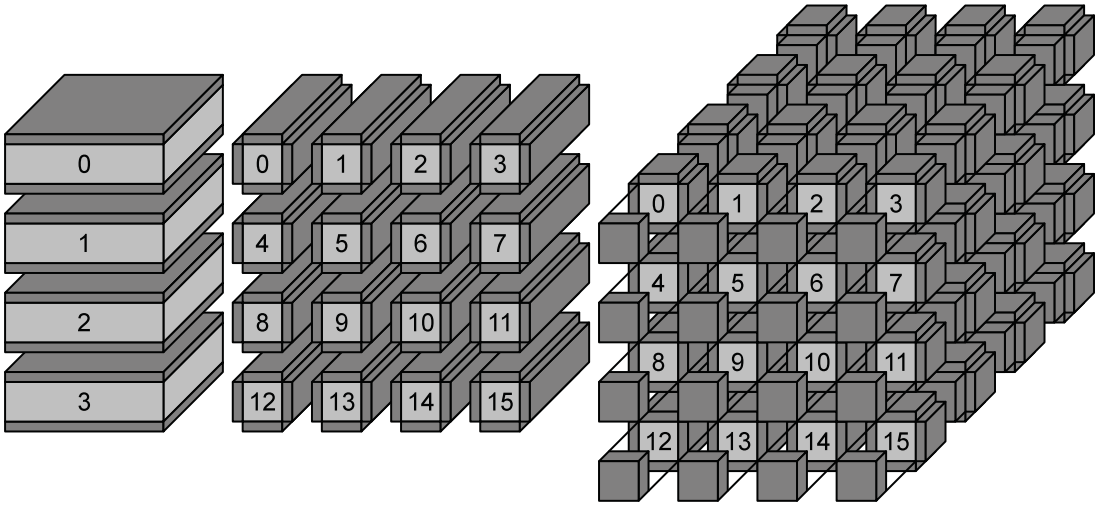


Figure 5.2: Three rectilinear three-dimensional lattice decomposition methods. In one-dimension (left), two-dimensions (centre) and three-dimensions (right). The dark grey cells represent the neighbouring cells of the lattice to which each section must have access.

parallel sections.

5.6 Multi-Core - POSIX Threads

To decompose a finite-differencing simulation across multiple CPU cores with Pthreads, separate threads are launched to update separate sections of the field. The field should be partitioned to evenly distribute the computational workload among all of the threads, and the number of threads should be configured such that they make full use of the CPU cores without interfering with each other.

For each time step the threads will iterate over their section of the field and compute the new value of the field after the time step. The threads must synchronise with each other after every integration stage and time step to ensure that all the data is consistent. Semaphores are used for this purpose. After each thread has performed its computational task it will signal semaphores to inform the neighbouring threads that it has finished. It will then wait for all neighbours to signal that they have completed before continuing onto the next step. The number of neighbours and semaphores will depend on the decomposition method used.

5.6.1 Runge-Kutta 2^{nd}

An example Pthreads implementation of a two-dimensional finite-differencing simulation using the RK2 method is shown in Listing 5.11. The lattice is split into `NUM_THREADS` sections using the one-dimensional decomposition method discussed in Section 5.5. For each time step the threads will process their section of the field and synchronise with their neighbouring threads. In this example

each thread will have two neighbours with which to synchronise because it uses the one-dimensional decomposition method.

To compute the Runge-Kutta 2^{nd} order integration method, two synchronisations per step are required. This integration method requires an intermediate lattice to be computed which is then used for the computation of the final system. As the cells on the borders of each section require values from the intermediate stage of the neighbouring section, the threads must synchronise to ensure these intermediate values are correct.

These two synchronisations can be seen in the RK2 code shown in Listing 5.11. For each step the threads will compute the midpoint step with $u1 = u0 + f(u0) * \frac{h}{2}$, synchronise with the other threads and then compute the final system using $u2 = u0 + f(u1) * h$. After this system is computed the threads will synchronise again to ensure all threads have completed their computation before continuing onto the next time step.

Listing 5.11: Parallel implementation of the RK2 integration method using the Pthreads multi-threading library.

```
void* rk2_pthreads(void *threadid) {
   ...
   int YN = Y/NUM_THREADS;
   for(int t = 0; t < no_steps; t++) {
      for(int iy = id*YN; y < (id+1)*YN; iy++) {
         for(int ix = 0; ix < X; ix++) {
            // u1 = u0 + f(u0) * h/2
         }
      }
      sem_post(&sem_m1[id]);
      sem_post(&sem_p1[id]);
      sem_wait(&sem_m1[idm1]);
      sem_wait(&sem_p1[idp1]);

      for(int iy = id*YN; y < (id+1)*YN; iy++) {
         for(int ix = 0; ix < X; ix++) {
            // u2 = u0 + f(u1) * h
         }
      }
      sem_post(&sem_m1[id]);
      sem_post(&sem_p1[id]);
      sem_wait(&sem_m1[idm1]);
      sem_wait(&sem_p1[idp1]);
   }
   sem_post(&s_stop[id]);
}

int main(int argc, char** argv) {
   u0 = new double[Y*X];
   u1 = new double[Y*X];
   ...
   for(int i = 0; i < NUM_THREADS; i++){
      pthread_create(&threads[i], NULL, rk2_pthreads, (void *)i);
   }
   for(int i = 0; i < NUM_THREADS; i++) {
      sem_wait(&s_stop[i]);
   }
}
```

In this example the main function simply creates the threads and waits for them to signal their completion before returning. There are many other ways of implementing such a simulation using Pthreads. This example is simply designed to give an idea of the low-level thread sychronisation required.

5.7 Multi-Core -Threading Building Blocks

TBB makes it easy to implement a parallel finite-differencing simulation for a multi-core CPU. The update method can be parallelised by performing the iteration over the lattice in parallel using the TBB method `parallel_for`. TBB will iterate over the data-set in parallel by partitioning the iteration range into segments and distributing these iteration segments across the cores of the processor.

The total range of iteration (the size of the lattice) and the size of the segments can be defined by the user with either `blocked_range`, `blocked_range2d` or `blocked_range3d` depending on the dimensionality of the lattice. These structures define how TBB should partition the lattice into segments to be processed separately. Each of these methods represents one of the lattice decomposition methods described in Section 5.5. If a lattice is four-dimensional (or higher) it must be partitioned using a lower-dimensionality `blocked_range` object as TBB does not provide a four- or higher-dimensional implementation of **blocked_range**.

5.7.1 Runge-Kutta 2^{nd}

Listing 5.12 contains a code snippet showing how TBB can make use of a multi-core CPU to compute a two-dimensional RK2 finite-differencing update method. This implementation will split the lattice into segments of size (64,64) in a two-dimensional decomposition method as defined by the `blocked_range2d` object. These segments will be distributed between the cores on the processor and will be updated using the RK2 method.

All task distribution, load balancing and synchronisation will be performed by TBB automatically. The implementation parallelises the computation without the user having to create threads or explicitly define thread synchronisation. This code can be compared to the previous Pthreads implementation in Listing 5.11 to show the difference in code complexity between high- and low-level parallel libraries.

In order to compute the two stages of the RK2 method, two `parallel_for` calls are used. This is necessary because the first stage must complete before the second can start. While TBB can make parallel programming significantly easier, it still requires the developer to have some knowledge of parallel processing.

Listing 5.12: Threading Building Blocks implementation of the RK2 method. Two `parallel_for` are used to compute the two stages.

```
void rk2_tbb(double *u0, double *u1, double *u2) {
   parallel_for( blocked_range2d<size_t>(0, Y, 64, 0, X, 64),
   [=](blocked_range2d<size_t>& r) {
      int y_begin = r.rows().begin();
      int y_end = r.rows().end();
      int x_begin = r.cols().begin();
      int x_end = r.cols().end();
      for ( int iy = y_begin; iy != y_end; ++iy ) {
         for( int ix = x_begin; ix != x_end; ++ix ) {
            // u1 = u0 + f(u0) * h
         }
      }
   });
   parallel_for( blocked_range2d<size_t>(0, Y, 64, 0, X, 64),
   [=](blocked_range2d<size_t>& r) {
```

```
        int y_begin = r.rows().begin();
        int y_end = r.rows().end();
        int x_begin = r.cols().begin();
        int x_end = r.cols().end();
        for ( int iy = y_begin; iy != y_end; ++iy ) {
           for( int ix = x_begin; ix != x_end; ++ix ) {
              // u2 = u0 + f(u1) * h
           }
        }
     });
}
```

The example uses a lambda function as the update method of the `parallel_for` call identified by the preceding `[=]`. Lambda functions are defined in the C++0x specification [143]. It can also be implemented easily in C++ using the current standard [144] by using a class with an overloaded `operator()`, however it does result in more cluttered code.

5.8 Cluster - MPI

MPI can be used to allow a cluster of machines to collaboratively compute a single finite-differencing simulation. For this type of distributed memory machine, the storage of the lattice must be split across the cluster nodes using a suitable parallel decomposition method. Each node is responsible for computing the simulation on the section of lattice assigned to it. For every stage of the simulation, each node must communicate the bordering sections of its lattice with the neighbouring nodes. The number of neighbours will depend on the parallel decomposition method used to split the lattice between the nodes.

The cluster nodes can use MPI functions to identify themselves, work out which nodes are neighbours, communicate data with and synchronise with these neighbours. There are two main methods for MPI communication - blocking and non-blocking. When blocking communication is used, the node will either send or receive data from another node and wait until the communication is completed before continuing. Non-blocking communication allows the node to continue computation while waiting for the data to be communicated across the interconnect network. This allows the communication and computation to be overlapped which is designed to reduce overall execution time.

5.8.1 Runge-Kutta 2^{nd}

In order to compute a finite-difference simulation using the RK2 integration method, two sets of communication must be performed. Initially the intermediate stage will be computed and the borders of the lattice will be exchanged with the neighbour nodes. Using this information the final lattice can then be computed and again the borders of the lattice must be communicated with the neighbours. The bordering area and number of neighbours will depend on the parallel decomposition method. In the example in Listing 5.13 a one-dimensional decomposition method is used. This means there are two border areas and each node will have two neighbours with which to exchange data.

In order to compute the simulation for its section of the lattice, each node must actually store a slightly larger lattice which contains the bordering cells belonging to neighbouring nodes. In the

example in Listing 5.13 the lattice that each cell stores is actually of size [(Y/P+2*H)][X]. The decomposition method splits the lattice in the Y dimension so each lattice segment is of width X while the height of the lattice segment is Y/P (where P is the number of nodes) plus the two border segments of height H. The variable H represents the memory halo or maximum stencil size of the equation. Larger stencils mean larger border areas and thus more communication. The Cahn-Hilliard equation contains the biharmonic stencil and has a memory halo of H=2.

By performing the computation in several parts, the time spent waiting for the communication to complete can be reduced. The first parts compute the bordering cells - in this example there are two border areas. These border areas are then exchanged with the neighbours using the non-blocking communication calls `MPI_Isend` and `MPI_Irecv`. While this data is being communicated across the network, the node can continue computing the simulation for the rest of the lattice. This method of overlapping computation and communication using the RK2 integration method and a one-dimensional decomposition method is shown in Listing 5.13.

Listing 5.13: MPI implementation of the RK2 method using a one-dimensional decomposition method. Communication between the nodes is performed using the MPI `MPI_Isend`, `MPI_Irecv` and `MPI_Waitall` commands.

```
void cahn_hilliard_rk2(float *u0, float *u1, float *u2) {
    int YN = Y/P;
    MPI_Irecv(&u1[(YN+H)*X], X*H, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[0]);
    MPI_Irecv(&u1[0],        X*H, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[1]);
    for(int iy = HALO; iy < 2*H; iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u1 = u0 + f(u0) * h/2
        }
    }
    MPI_Isend(&u1[H*X], X*H, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[2]);
    for(int iy = YN; iy < (YN+H); iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u1 = u0 + f(u0) * h/2
        }
    }
    MPI_Isend(&u1[YN*X], X*H, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[3]);
    for(int iy = 2*H; iy < YN; iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u1 = u0 + f(u0) * h/2
        }
    }
    MPI_Waitall(4, request, status);

    MPI_Irecv(&u2[(YN+H)*X], X*H, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[0]);
    MPI_Irecv(&u2[0],        X*H, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[1]);
    for(int iy = H; iy < 2*H; iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u2 = u0 + f(u1) * h
        }
    }
    MPI_Isend(&u2[H*X], X*H, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[2]);
    for(int iy = YN; iy < (YN+H); iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u2 = u0 + f(u1) * h
        }
    }
    MPI_Isend(&u2[YN*X], X*H, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[3]);
    for(int iy = 2*H; iy < YN; iy++) {
        for(int ix = 0; ix < X; ix++) {
            // u2 = u0 + f(u1) * h
        }
    }
    MPI_Waitall(4, requests, statuses);
}
```

The one-dimensional decomposition method is used as the example because of the code simplicity. However, it will not scale well to a large number of nodes due to the large border area per lattice segment. A higher dimensionality decomposition method would improve the scalability of the MPI implementation to the large number of nodes commonly used for such implementations. There will be more neighbours with which to communicate but overall there will be less data to communicate per node.

5.9 Graphical Processing Units - CUDA

GPU devices were originally designed for computing the three-dimensional rendering pipeline. For this reason they are highly optimised for performing computations on images, which are essentially two-dimensional lattices. GPUs are well suited to computing finite-differencing simulations efficiently because this type of computation is very similar to the image-processing for which they were originally intended.

In order to perform a finite-differencing update on a graphics card, the computational problem must be decomposed into many threads that can be executed on the GPU. Rather than using a coarse-grained field decomposition method and distributing the sections of the field between the GPU cores, one thread is created for each lattice cell. This method of decomposing the lattice is essentially the same as using a parallel decomposition method with the same dimensionality as the lattice and a grain size of one. Each cell of the lattice becomes its own segment which will be processed by a different thread.

In almost all cases this will result in many more threads than there are cores on the GPU.These threads will be managed by the GPU hardware and executed on the multiprocessors. The CUDA Programming Guide [103] recommends that any program with at least 1000 blocks of threads should scale well to the next several generations of GPU hardware.

The other major consideration of programming a finite-differencing simulation on a GPU is the memory access. Because the GPU cannot access system memory (at least not efficiently [145]) the input data must be loaded into the GPU and the results subsequently copied back to the host after the computation is complete. All of this input/output must be performed using the GPU's `global` memory as this is the only memory type that can be accessed by the host CPU.

It is beneficial for the data to remain purely on the GPU during computation as copying data backwards and forwards between the host wastes time. For most finite-differencing simulations it is possible for the data to remain on the GPU until the end of the simulation and only the final result is copied back to the host. It may be necessary to copy the data to the host during computation for analysis, however many common analysis algorithms such as Connected-Component Labeling can be efficiently implemented on the GPU [146].

5.9.1 Runge-Kutta 2^{nd} Order

In order to implement the RK2 method with CUDA, two separate kernels are required. Because there is no way to synchronise between thread blocks during a kernel's execution, the computation of each

stage must be split into a separate kernel. The intermediate system state is computed by the first kernel `rk2a_cuda` and `cudaThreadSynchronize` is used to ensure that this kernel is complete before the second kernel is launched. This second kernel `rk2b_cuda` will use the intermediate system state to compute the final value. This code can be seen in Listing 5.14.

Listing 5.14: CUDA program to compute a finite-differencing simulation using the Runge-Kutta 2^{nd} order integration method.

```
__global__ void rk2a_cuda(float* u0, float* u1) {
        int ix = (blockIdx.x * blockDim.x) + threadIdx.x;
        int iy = (blockIdx.y * blockDim.y) + threadIdx.y;

        // u1 = u0 + f(u0) * h/2
}

__global__ void rk2b_cuda(float* u0, float* u1, float *u2) {
        int ix = (blockIdx.x * blockDim.x) + threadIdx.x;
        int iy = (blockIdx.y * blockDim.y) + threadIdx.y;

        // u2 = u0 + f(u1) * h
}

int main(int argc, char** argv) {
        host_u0 = new float[Y*X];
        float *u0, *u1, *u2;
        cudaMalloc( (void**) &u0,  Y*X*sizeof(float));
        cudaMalloc( (void**) &u1,  Y*X*sizeof(float));
        cudaMalloc( (void**) &u2,  Y*X*sizeof(float));
        ...
        cudaMemcpy(u0, host_u0, Y*X*sizeof(float), cudaMemcpyHostToDevice);
        dim3  threads(16,    16);
        dim3  blocks(  X/16, Y/16);
        for(int t = 0; t < no_steps; t++) {
                rk2a_cuda <<< blocks, threads >>> (u0, u1);
                cudaThreadSynchronize();
                rk2b_cuda <<< blocks, threads >>> (u0, u1, u2);
                cudaThreadSynchronize();
                swap(&u0, &u1);
        }
        cudaMemcpy(host_u0, u0, Y*X*sizeof(float), cudaMemcpyDeviceToHost);
}
```

Extending this program design to a higher-order integration method is a relatively simple process. Each stage of the integration method will require a kernel to compute it. The main performance consideration for these kernels is how they access data from memory.

5.9.2 GPU Memory Access

GPUs contain many memory types which must be accessed explicitly by the programmer, however Fermi architecture GPUs do support automatically-cached access to global memory. All the input from the host must be copied into global memory and the output copied back out. However, there are several ways in which that data can be accessed by the kernels during computation. Every kernel can either read all the data directly from global memory or use shared memory or the texture cache. These three methods of accessing data have been implemented and compared:

5.9.3 Global Memory

The easiest way to implement the kernel is to simply access all required data directly from global memory. Each kernel must read its lattice cell value and any neighbouring values required by the

equation, compute the equation and then write the output value to another output array. The number of neighbouring values that must be read will depend on the stencils used by the equation.

The performance of the kernel will depend on the shape of the thread blocks used. Thread blocks should be either strips or rectangles of threads with the width as a multiple of sixteen for Tesla architecture GPUs and a multiple of thirty-two for Fermi GPUs. This is to ensure that global memory transactions are coalesced. When sixteen/thirty-two sequential and aligned threads access sixteen/thirty-two sequential and aligned memory addresses, the transactions will be coalesced into a single access. The performance of this kernel greatly depends on this coalesced memory access. Listing 5.15 shows a code snippet of the nearest neighbours of a lattice cell being read from global memory.

Listing 5.15: CUDA kernel that loads neighbouring values from a lattice **u0** stored in global memory.

```
int x = (blockIdx.x * blockDim.x) + threadIdx.x;
int y = (blockIdx.y * blockDim.y) + threadIdx.y;

float u_ym1x = u0[(y-1) * X +  x   ];
float u_yxm1 = u0[ y    * X + (x-1)];
float u_yx   = u0[ y    * X +  x   ];
float u_yxp1 = u0[ y    * X + (x+1)];
float u_yp1x = u0[(y+1) * X +  x   ];
```

Because the threads must access neighbouring cells, neighbouring threads (threads in the same thread block) must often access the same lattice cells from global memory. This introduces an element of duplicated work as values in global memory will be fetched multiple times by threads in the same block. Fermi cards have L1/L2 cache which will be automatically used when the kernels access global memory. The cache will reduce the number of global memory accesses required as a value fetched from global memory will be stored in the cache and any subsequent threads can simply read it from the cache [122]. However, Tesla architecture GPUs do not have this capability and either shared memory or the texture cache must be used to cache the data.

5.9.4 Shared Memory

One option is to make use of shared memory to reduce duplicated global memory access. Shared memory is a type of on-chip memory that all threads within the same block can use to share data. Because threads within the same block must often access the same values from memory as their neighbours, they can use this shared memory to reduce duplicated work. This is effectively using the shared memory as a cache for global memory access.

However, this does raise the question of which threads should fetch which values from global memory and save them in shared memory. It is sensible that each thread fetch the value of the cell it updates and loads it into shared memory. However, the threads on the border of the block will require values from memory that are used by no other thread in that block.

There are two obvious options for overcoming this issue. The first is for each thread on the border of a block to load any possible values from shared memory and the remaining values from global memory. The disadvantage of this approach is that the block can only complete once all threads are completed. Any threads within the same warp as a border thread must wait for the border thread's

global memory transactions to complete before continuing. This imbalance can cause a significant degradation of performance, especially for equations with large stencils as bordering cells may have to load many values from global memory.

A better option is to create extra threads whose sole purpose is to read data from global memory into the shared memory cache. These extra threads overlap with threads for other thread blocks but remove the need for bordering cells to load data from global memory. The number of global memory transactions is not reduced, but they are more evenly distributed between the threads in the block. The kernel code for this implementation can be seen in Listing 5.16.

Listing 5.16: CUDA kernel that uses shared memory as a cache for accessing neighbouring values in a lattice **u0** stored in global memory. Extra border threads are used to read values from global memory into the shared cache.

```
int x = (blockIdx.x * INNER_BLOCK_SIZE_X) + threadIdx.x;
int y = (blockIdx.y * INNER_BLOCK_SIZE_Y) + threadIdx.y;

int tx = threadIdx.x;
int ty = threadIdx.y;

cache[ty*blockDimx + tx] = u0[y * X + x];

__syncthreads();

float u_ym1x = u0[(y-1) * X +  x   ];
float u_yxm1 = u0[ y    * X + (x-1)];
float u_yx   = u0[ y    * X +  x   ];
float u_yxp1 = u0[ y    * X + (x+1)];
float u_yp1x = u0[(y+1) * X +  x   ];
```

This implementation is more important for the Tesla than the Fermi architecture GPUs. Because the Fermi GPUs have automatic L1/L2 cache, global memory access is already cached and the shared memory is effectively duplicating this cache. The Tesla cards do not have any global memory cache and thus it can be expected that using shared memory as a cache will have a greater impact on performance.

5.9.5 Texture Memory

Another option for cached memory access is the texture cache. Texture memory was designed for rendering textured objects. Texture memory is optimised for loading values from memory stored in the same spatial locality. This is a similar memory access pattern as accessing neighbouring values for finite-differencing simulations.

Because the memory access patterns are very similar to the original purpose that the texture cache was designed for, it can be expected that the texture cache will provide good performance on Tesla GPUs. However, for Fermi GPUs it is recommended that unless features of the texture processing units such as the address calculations and filtering are used then cached global memory should be used instead. This is because of the L1/L2 caches on the Fermi cards which have a higher bandwidth than the texture cache.

The texture cache can be used by copying a lattice into a `cudaArray` and binding it to a texture. CUDA provides the mechanism for one-, two- or three-dimensional textures. When a value is retrieved from a texture, the value will be read from global memory only if it is not already present

in the texture cache. The texture cache performs best when threads within the same warp access addresses that are near each other [103]. Listing 5.17 shows the CUDA kernel reading values from a two-dimensional texture `texture_u0`.

Listing 5.17: CUDA kernel that uses texture memory to access neighbouring values from the two-dimensional texture **texture_u0**.

```
int x = (blockIdx.x * blockDim.x) + threadIdx.x;
int y = (blockIdx.y * blockDim.y) + threadIdx.y;

float u_ym1x = tex2D(texture_u0, x,   ym1);
float u_yxm1 = tex2D(texture_u0, xm1, y   );
float u_yx   = tex2D(texture_u0, x,   y   );
float u_yxp1 = tex2D(texture_u0, xp1, y   );
float u_yp1x = tex2D(texture_u0, x,   yp1);
```

One downside of using the texture cache is that kernels cannot write to a `cudaArray`. This means that the kernels must write to another part of global memory and the host must copy this output back into a `cudaArray` after the kernel has completed. The extra memory copy can have a significant impact on performance and in some cases negate the benefits the texture cache provides.

5.10 Multi-GPU - CUDA & Pthreads

GPUs are powerful parallel computers that can be programmed relatively easily with CUDA; however, a simple CUDA program cannot be scaled beyond a single GPU. To extend a CUDA program, multiple host threads must be used to connect multiple GPU devices independently. This can allow GPU programs to make use of as many GPU devices as the host can support.

To simulate a finite-differencing simulation on multiple GPUs, the lattice must be split between the devices. Any of the decomposition methods discussed in Section 5.5 can be used but each method will have different implications on performance. The borders of each section must be communicated between GPU devices after each stage of computation.

There have been several versions of CUDA released during the period that this research has been conducted. For all CUDA versions up to and including CUDA 3.2, there has been no direct communication possible between GPUs. Any data that must be transferred from one GPU to another has had to be copied through the host memory. To move data between two GPUs, the controlling thread of the source GPU must copy the data out of the device memory and into host memory. It can then signal the thread controlling the destination GPU that the data is ready and this thread can then copy it into the destination GPU.

However, the release of CUDA 4.0 in May 2011 has made it possible for GPUs to communicate directly. Fermi GPUs with compute capability 2.0 can access data in another GPU's memory and the host can directly transfer data from one GPU to another. This makes multi-GPU programming significantly easier as it does not require the data to be copied into host memory.

For the multi-GPU implementation example, the lattice is decomposed in only the highest dimension. The advantage of this is that the bordering information will be stored in contiguous memory and can be copied in and out of a GPU device with a single `cudaMemcpy` call. A multi-dimensional

decomposition method will have better scalability but will require more complex border communication to more neighbouring devices. As multi-GPU algorithms are generally limited to a small number of GPUs, the one-dimensional decomposition method is usually suitable.

Two implementations are shown for performing a finite-differencing update on multiple GPUs which do not support direct GPU communication. Both implementations use CUDA to perform the update and Pthreads for the CPU threads. The code examples show the algorithm for computing a single stage of the RK2 integration method. The two implementations differ in their use of synchronous and asynchronous data transfer.

5.10.1 Synchronous Communication

The synchronous communication method is the easiest implementation of a multi-GPU finite-difference simulation. This implementation uses `cudaMemcpy` to communicate bordering data between the host threads and the GPU devices in a synchronous fashion.

Each thread will call a CUDA kernel to compute one stage of computation on its section of the lattice. Once this is completed it will copy the necessary border information of the lattice into a swap space in the host memory. After synchronising with the neighbouring host threads it will then copy the new bordering data it requires into GPU memory. An example of this implementation using the RK2 method and a one-dimensional lattice decomposition method is shown in Listing 5.18.

Listing 5.18: Multi-GPU implementation of the RK2 method using synchronous communication.

```
void sync(int id) {
        int idm1 = (id == 0) ? NUM_THREADS-1 : id-1;
        int idp1 = (id == NUM_THREADS-1) ? 0 : id+1;
        sem_post(&sem_m1[id]);
        sem_post(&sem_p1[id]);
        sem_wait(&sem_m1[idp1]);
        sem_wait(&sem_p1[idm1]);
}

for(int i = 0; i < no_steps; i++) {
   rk2_a<<< grid, threads >>>(u0, u1, u2, h);
   cudaThreadSynchronize();
   cudaMemcpy(&sw[id][0],   &u1[H*X],  H*X*sizeof(float), cudaMemcpyDeviceToHost);
   cudaMemcpy(&sw[id][H*X], &u1[YN*X], H*X*sizeof(float), cudaMemcpyDeviceToHost);
   sync(id);
   cudaMemcpy(&u1[(YN+H)*X], &sw[idp1][0],   H*X*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(&u1[0],        &sw[idm1][H*X], H*X*sizeof(float), cudaMemcpyHostToDevice);
   sync(id);

   rk2_b<<< grid, threads >>>(u0, u1, u2, h);
   cudaThreadSynchronize();
   cudaMemcpy(&sw[id][0],   &u2[H*X],  H*X*sizeof(float), cudaMemcpyDeviceToHost);
   cudaMemcpy(&sw[id][H*X], &u2[YN*X], H*X*sizeof(float), cudaMemcpyDeviceToHost);
   sync(id);
   cudaMemcpy(&u2[(YN+H)*X], &sw[idp1][0],   H*X*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(&u2[0],        &sw[idm1][H*X], H*X*sizeof(float), cudaMemcpyHostToDevice);
   sync(id);
   swap(&u0, &u2);
}
```

This method is simple to implement and provides an easy way to utilise multiple GPUs for a single simulation. However, the downside is that each GPU must calculate the update for the lattice then stop and sit idle while the CPU threads exchange data. When the lattice size and memory halo are both small, this memory exchange does not take long and it does not have a large impact on

performance. However, as the lattice size and/or memory halo increases, the exchange will cause a greater performance loss. Any time the GPU sits idle is a waste of resources. To minimise this idle-time it is possible to make use of the asynchronous memory copy/execution capabilities of the GPU.

5.10.2 Asynchronous Communication

The asynchronous communication implementation makes use of the CUDA asynchronous memory/execution feature to reduce idle GPU time. CUDA supports asynchronous host-device memory access and execution for GPUs with compute capability 1.1 or higher [31]. Device kernels and host/device memory transfers in separate streams can be performed at the same time. This feature can be used to communicate the necessary border data between the host and the GPU while the device is still computing the simulation.

For this implementation to work, the computation must be split into separate streams such that the border cells are computed first and can then be copied to the host. Initially the update for bordering lattice cells is computed. Once this is completed an asynchronous memory copy will be used to start copying the data to the host. At the same time the kernels to update the rest of the lattice are launched. While these kernels are computing, the host threads will wait for memory copy to complete, synchronise with the other threads and then copy the new borders into the GPU using another asynchronous memory copy. An example of this implementation using a one-dimensional lattice decomposition method and the RK2 integration method is shown in Listing 5.19.

Listing 5.19: Implementation of the Asynchronous Copy multi-GPU implementation.

```
for(int i = 0; i < no_steps; i++) {
   rk2_a_border <<< grid_border,  block_border,  0, stream_border >>>(u0, u1, u2, h);
   cudaStreamSynchronize(stream_border);
   rk2_a     <<< grid_compute, block_compute, 0, stream_compute>>>(u0, u1, u2, h);
   sem_wait(&sem_recv_m1[idp1]);
   sem_wait(&sem_recv_p1[idm1]);
   cudaMemcpyAsync(&sw[id][0],   &u1[H*X],  H*X*sizeof(float),
                   cudaMemcpyDeviceToHost, stream_border);
   cudaMemcpyAsync(&sw[id][H*X], &u1[YN*X], H*X*sizeof(float),
                   cudaMemcpyDeviceToHost, stream_border);
   cudaStreamSynchronize(stream_border);
   sem_post(&sem_send_m1[id]);
   sem_post(&sem_send_p1[id]);
   sem_wait(&sem_send_m1[idp1]);
   sem_wait(&sem_send_p1[idm1]);
   cudaMemcpyAsync(&u1[(YN+H)*X], &sw[idp1][0],   H*X*sizeof(float),
                           cudaMemcpyHostToDevice, stream_border);
   cudaMemcpyAsync(&u1[0],         &sw[idm1][H*X], H*X*sizeof(float),
                             cudaMemcpyHostToDevice, stream_border);
   sem_post(&sem_recv_m1[id]);
   sem_post(&sem_recv_p1[id]);
   cudaStreamSynchronize(stream_compute);
   cudaStreamSynchronize(stream_border);

   //-------------------------------------------------------------------------------------------------------
   rk2_b_border<<< grid_border,  block_border,  0, stream_border >>>(u0, u1, u2, h);
   cudaStreamSynchronize(stream_border);
   rk2_b       <<< grid_compute, block_compute, 0, stream_compute>>>(u0, u1, u2, h);
   sem_wait(&sem_recv_m1[idp1]);
   sem_wait(&sem_recv_p1[idm1]);
   cudaMemcpyAsync(&sw[id][0],    &u2[H*X], H*X*sizeof(float),
                   cudaMemcpyDeviceToHost, stream_border);
   cudaMemcpyAsync(&sw[id][H*X], &u2[YN*X], H*X*sizeof(float),
                   cudaMemcpyDeviceToHost, stream_border);
```

```
        cudaStreamSynchronize(stream_border);
    sem_post(&sem_send_m1[id]);
    sem_post(&sem_send_p1[id]);
    sem_wait(&sem_send_m1[idp1]);
    sem_wait(&sem_send_p1[idm1]);
    cudaMemcpyAsync(&u2[(YN+H)*X], &sw[idp1][0],   H*X*sizeof(float),
                                cudaMemcpyHostToDevice, stream_border);
    cudaMemcpyAsync(&u2[0],         &sw[idm1][H*X], H*X*sizeof(float),
                                cudaMemcpyHostToDevice, stream_border);
    sem_post(&sem_recv_m1[id]);
    sem_post(&sem_recv_p1[id]);
    cudaStreamSynchronize(stream_compute);
    cudaStreamSynchronize(stream_border);

    swap(&u0, &u2);
}
```

There are some programatic challenges that must be overcome when implementing this design that are worth noting. For the GPU to copy data from the device memory to the host memory asynchronously, the host memory must be allocated by CUDA to ensure that it is page locked. This is normally performed using the CUDA function `cudaMallocHost( void **ptr, size_t size)`. However, memory declared using this function will only be accessible to the thread that declares it. This means that exchanging the bordering cells requires an extra CPU memory copy to copy the data between the page-locked memory for each thread.

This can be overcome by allocating the memory using the function `cudaHostAlloc(void **ptr, size_t size, unsigned int flags)` with the flag `cudaHostAllocPortable`. This flag tells the compiler to make the memory available to all CUDA contexts rather than only the one used to declare it [103]. In this way both threads can use the memory to exchange border cells.

5.10.3 Direct Communication

At the time of writing, CUDA 4.0 with support for direct GPU-GPU communication is still in development. This CUDA version allows for direct communication of data from one GPU device to another. This can take two forms - direct copy or direct access [123].

Direct access allows the threads executing on one GPU to access data values stored in the other GPU's memory. This means that the communication of data is performed during the computation of the simulation. The two GPU devices must still be synchronised with each other to ensure the values the threads are fetching are correct.

Direct copy allows the host CPU to copy data directly from one GPU to another without having to first be transferred to host memory. This should make the communication of border data more efficient and provide a better computation to communication ratio.

As CUDA 4.0 has not yet been released, implementations using this direct GPU communication have not been tested. Once CUDA 4.0 has been released the direct GPU communication will only be available on GPUs with compute capability 2.0 and higher. The asynchronous copy implementation can be executed on any GPU with compute capability 1.1 and higher and is thus the more flexible implementation.

5.11 GPU Cluster - CUDA & MPI

Decomposing a finite-differencing simulation across a GPU cluster is a similar problem to splitting it across multiple GPUs within the same host. The main difference is the communication between host threads. Rather than simply communicating the data through a swap area of memory, the bordering data must be communicated across a network interconnect. MPI is used to execute the program on a cluster and manage the communication between the nodes. CUDA is used to connect to the GPUs and launch kernels.

Two implementations of a GPU cluster finite-differencing update method are discussed, both of which make use of the asynchronous memory copy/execution features of the GPU. The implementations differ in the manner with which the nodes communicate.

5.11.1 Asynchronous Communication

The asynchronous communication implementation is effectively the multi-GPU asynchronous memory copy implementation from Section 5.10.2 adapted for a cluster. Instead of exchanging borders with another thread through the host memory, the data is sent over the network using MPI calls. With this communication method, each node will exchange border information with all neighbours as determined by the lattice decomposition method used. This implementation can be used with any of the decomposition methods discussed in Section 5.5. Listing 5.20 shows the main update loop of this implementation for the RK2 method and one-dimensional lattice decomposition.

Listing 5.20: Implementation of the Asynchronous Copy multi-GPU implementation.

```
for(int i = 0; i < no_steps; i++) {
    MPI_Irecv(&recv[0],    H*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &recv_requests[0]);
    MPI_Irecv(&recv[H*X], H*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &recv_requests[1]);
    rk2a_border <<< grid_border,  block_border,  0, stream_border  >>>(u0, u1, u2, h);
    rk2a        <<< grid_compute, block_compute, 0, stream_compute >>>(u0, u1, u2, h);
    cudaStreamSynchronize(stream_border);
    cudaMemcpyAsync(&send[0],    &u1[H*X],  H*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream_border);
    cudaMemcpyAsync(&send[H*X], &u1[YN*X], H*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream_border);
    MPI_Isend(&send[0],    H*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &send_requests[0]);
    MPI_Isend(&send[H*X], H*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &send_requests[1]);
    MPI_Waitall(2, recv_requests, statuses);
    cudaMemcpyAsync(&u1[0],         &recv[0],   H*X*sizeof(float),
                    cudaMemcpyHostToDevice, stream_border);
    cudaMemcpyAsync(&u1[(YN+H)*X], &recv[H*X], H*X*sizeof(float),
                    cudaMemcpyHostToDevice, stream_border);
    MPI_Waitall(2, send_requests, statuses);
    cudaStreamSynchronize(stream_border);
    cudaStreamSynchronize(stream_compute);

    MPI_Irecv(&recv[0],    H*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &recv_requests[0]);
    MPI_Irecv(&recv[H*X], H*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &recv_requests[1]);
    rk2b_border <<< grid_border,  block_border,  0, stream_border  >>>(u0, u1, u2, h);
    rk2b        <<< grid_compute, block_compute, 0, stream_compute >>>(u0, u1, u2, h);
    cudaStreamSynchronize(stream_border);
    cudaMemcpyAsync(&send[0],    &u2[H*X],  H*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream_border);
    cudaMemcpyAsync(&send[H*X], &u2[YN*X], H*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream_border);
    MPI_Isend(&send[0],    H*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &send_requests[0]);
    MPI_Isend(&send[H*X], H*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &send_requests[1]);
    MPI_Waitall(2, recv_requests, statuses);
    cudaMemcpyAsync(&u2[0],         &recv[0],   H*X*sizeof(float),
```

```
                        cudaMemcpyHostToDevice, stream_border);
    cudaMemcpyAsync(&u2[(YN+H)*X], &recv[H*X], H*X*sizeof(float),
                        cudaMemcpyHostToDevice, stream_border);
    MPI_Waitall(2, send_requests, statuses);
    cudaStreamSynchronize(stream_border);
    cudaStreamSynchronize(stream_compute);

    swap(&u0, &u2);
}
```

While this algorithm can provide reasonable performance for finite-differencing simulations on a GPU cluster [127] there is a large communication pipeline to send data between GPU devices. Each lattice border must be copied from the GPU to the host, sent to a neighbouring host using MPI and then copied into that node's GPU. One method to improve this communication has been described in [135].

5.11.2 Decoupled Communication

In "Overlapping Computation and Communication for Advection on Hybrid Parallel Computers" White and Dongarra have presented a method to improve this inter-node communication [135]. Rather than storing and computing the entire lattice on the GPU devices, the CPU stores and computes the simulation for a surrounding halo area. This halo area is the lattice section borders that must be communicated to the neighbouring nodes.

All bordering information which must be communicated to the neighbouring nodes is computed by the CPU rather than the GPU. This results in a border between the CPU and GPU sections as well as the borders between nodes. The CPU and GPU must communicate the bordering data between the CPU's halo and the GPU's section of the lattice. The advantage of this method is that it decouples the communication between GPUs. For each stage of the simulation, each CPU communicates with its GPU device and with the neighbouring nodes. No data must be transferred from GPU to another within a single computational stage. This method has been reported to significantly improve the performance of finite-differencing simulations on a GPU cluster [135].

5.12 Simulation Commonalities

Although these implementations use different languages, different decomposition methods and are designed for hardware architectures, they have a number of common elements. Identifying these elements is vital to constructing a generative programming system that can produce such parallel implementations.

All the implementations use the same basic concept of splitting the lattice into separate sections and assigning them to different threads. The different implementations generally use one of the parallel decomposition methods discussed in Section 5.5 and differ in terms of how coarsely the lattice is split. Many implementations can be configured to use any of these decomposition methods.

The other main element of these implementations is whether the computing architecture has a shared- or distributed-memory system. The multi-core CPU and single GPU implementations are shared memory architectures while the cluster, multi-GPU and GPU cluster implementation all have distributed memory.

For a shared memory system each thread simply needs to compute the simulation for its section and then synchronise with the other threads. For a distributed memory system the borders of the lattice must be communicated to the neighbouring nodes or devices. This required communication can be easily calculated from the stencils of the equation and the decomposition method used to split the lattice.

5.13 Conclusions

This chapter has shown how the numerical methods from Chapter 3 applied to the equations in Chapter 2 can be turned into code to compute the simulations. The method for constructing code that computes simulations for the Cahn-Hilliard, Lotka-Volterra and Ginzburg-Landau equations has been presented along with the algorithms to perform this in both sequential and parallel applications.

The algorithms presented here show how finite-difference solvers can be implemented for a variety of different parallel architectures. The implementation of parallel algorithms that make use of multiple parallel devices or a combination of different architectures has also be presented. The hardware architectures and parallel languages can be very different in nature and require different approaches and algorithms to obtain the optimal performance. However, despite these differences, a number of commonalities between the architectures and algorithms have been identified.

"However beautiful the strategy, you should occasionally look at the results."

Winston Churchill

6 Performance Results

6.1 Introduction

Performance results of the different parallel implementations have been collected to compare the algorithms, languages and architectures. Simulations of the three models discussed in Chapter 2 (Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra) have been constructed using the numerical methods described in Chapter 3. These simulations have been implemented for each of the different computing architectures discussed in Chapter 4 (single- and multi-core CPUs, GPUs and clusters) using the programming languages appropriate for each architecture and the parallel algorithms discussed in Chapter 5. These simulations also differ in terms of memory usage and the lattice decomposition method used.

The performance results of these different implementations are presented to compare the computational cost of simulating different models, to identify the optimal implementation for each architecture, to compare the performance of the different architectures and to show the importance of parallel computing for finite-difference simulations. The architectures/implementations are compared using a number of criteria - performance, maximum system size and scalability.

6.2 Model Computation

The first comparison drawn is the computational cost of simulating the three different models used as examples - the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations. These models have different computations and different memory access requirements. Variations are expected in the time taken to compute these models on the different architectures.

The parallel architectures discussed in this thesis are different configurations of two main processor types - CPUs and GPUs. The architectures discussed are different combinations of these processors on shared memory and distributed configurations. These two types of processing unit contain different optimisations for processing and memory access and it is expected that their performance will differ for the three models depending on their computational and memory access requirements.

Figure 6.1 shows a performance comparison of the Cahn-Hilliard (CH), Ginzburg-Landau (GL)

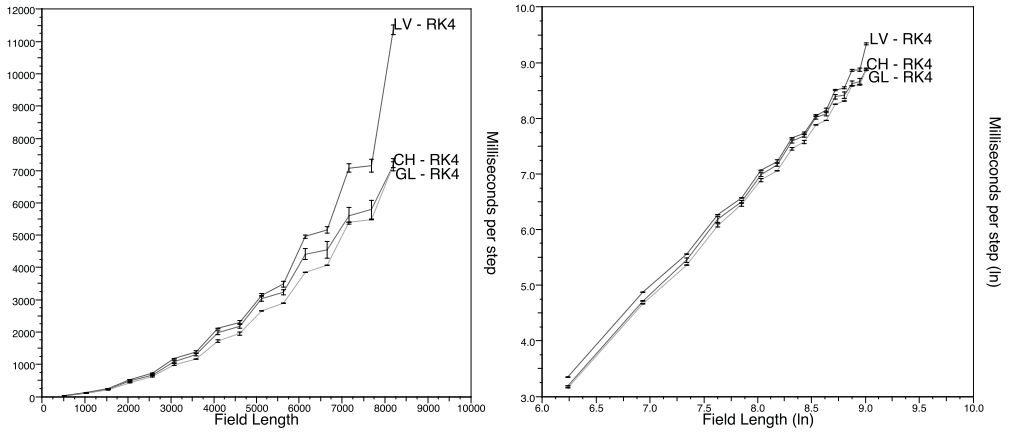


Figure 6.1: Comparison of the computational cost of simulation the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations on a CPU. Results shown in milliseconds per time step (left) and in ln-ln scale (right).

and Lotka-Volterra (LV) models using the RK4 integration method computed on a single-core CPU. It can be seen that the LV model takes significantly longer to compute than the CH and GL simulations, both of which have similar performance. One interesting feature of these plots is the kinks in the performance of all three models for lattice sizes which are not a power of two. It is believed that these performance variations are caused by effects of the CPU cache hierarchy. There is little that can be done about these performance variations as this cache cannot be explicitly controlled.

Figure 6.2 shows the computational cost of computing the three different models on a Fermi architecture GPU. These results show a somewhat different result to the CPU plot. On a GPU the GL and LV models have very similar computational cost while the CH model can be computed significantly faster than the others. The results are also a lot smoother as there are no uncontrolled caching effects.

For the other parallel implementations presented in the rest of this chapter, only the results for the Cahn-Hilliard model are given. The Ginzburg-Landau and Lotka-Volterra models show the same relative performance to the Cahn-Hilliard model for the different implementations and architectures. Presenting their results as well provides no new information and has thus been omitted.

6.3 Single-Core CPU

The first implementation tested is the single-threaded CPU implementation discussed in Chapter 5. This is a simple sequential algorithm and makes use of only one CPU core. However, this implementation does provide a useful reference point to which the other implementations can be compared. All speedup factors given in this chapter refer to the speedup of an implementation over this sequential program.

The machine used to test this implementation contains a 3.2 GHz Intel Core i7-970 with 12MB of

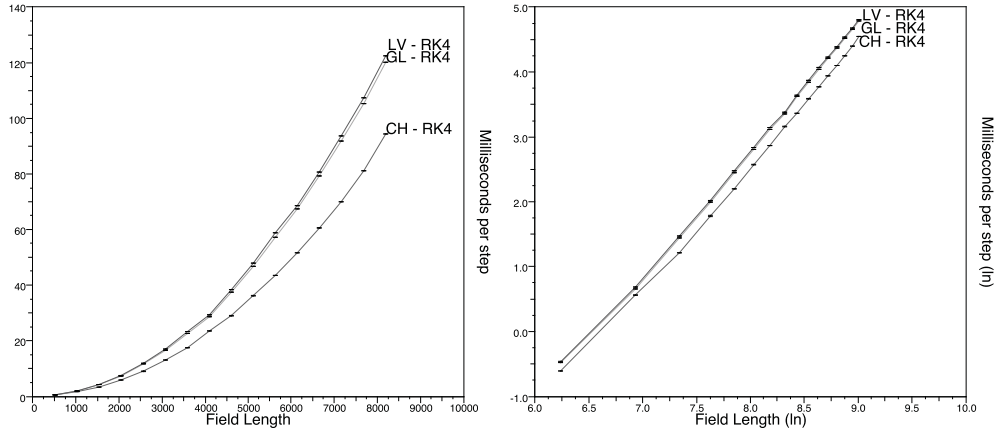


Figure 6.2: The cost of computing the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations on a GPU. All results are shown for the RK4 integration method. Plots show milliseconds per time step (left) and in ln-ln scale (right).

CPU cache and 12GB of DDR3-2000 RAM. The operating system running on this testing platform is Ubuntu 10.04 Lucid and the code compiled using gcc4.5 with optimisations (-O3). At the time of writing this CPU can be considered a high-performance processor.

Figure 6.3 shows the performance plots for the Cahn-Hilliard simulations executed on this platform using the Euler, RK2 and RK4 integration methods for system sizes of $N = \{512^2, 1024^2, 1536^2 ... 8192^2\}$.

These plots show that the simulation performance scales as expected. Execution time scales with N^d. Also as expected the two-stage RK2 method takes twice as long to compute as the one-stage Euler method, the four-stage RK4 method takes approximately four times as long. This simple implementation makes use of only one CPU core and does not utilise the full processing power of this CPU. To make full use of this multi-core CPU, a multi-threading library is required.

6.4 Multi-Core CPU

Simulations using two different multi-threading libraries have been implemented to make full use modern CPU's multiple cores. These are the Pthreads [110] and Intel's Threading Building Blocks [111] libraries discussed in Chapter 4. The simulation implementations that use these libraries are discussed in Chapter 5. These implementations have been tested and compared on the same machine used for the single-threaded sequential simulation.

6.4.1 Pthreads

The Pthreads simulation splits the lattice in one-dimension and each section is processed by a different thread. To determine the optimal number of threads to use, the simulation performance has

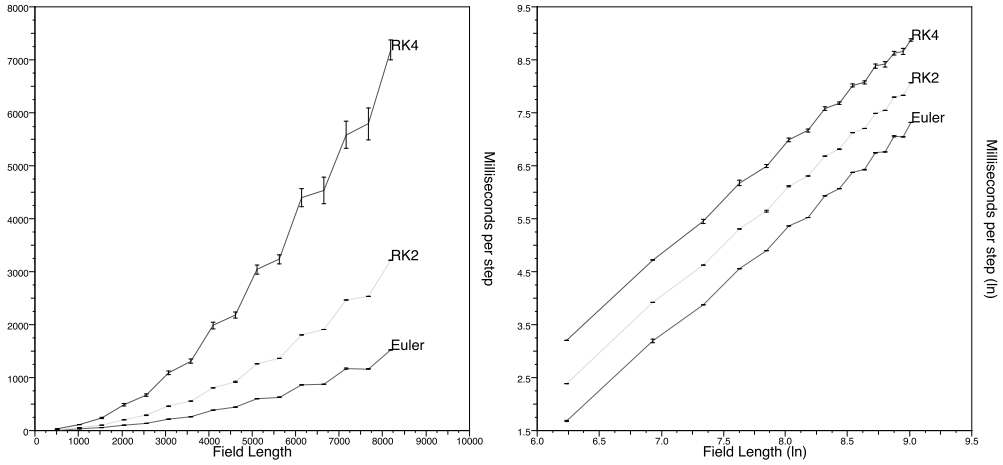


Figure 6.3: Performance results of the single-threaded CPU C implementations of the Cahn-Hilliard equation using the Euler, RK2 and RK4 integration methods. Results are shown in lattice length vs milliseconds per step (left) and in ln-ln scale (right).

been tested for a range of thread numbers $T = \{2, 4, 6...16\}$. Figure 6.4 shows the performance of the simulation for the different numbers of threads.

From this plot it can be seen that the Pthreads implementation offers the best performance when the computation is split between twelve threads. The Intel i7-970 processor contains six physical cores but supports twelve virtual cores. It can be seen from the plot that making use of these twelve virtual cores offers a significant performance benefit over using just six threads.

Figure 6.5 shows the performance results of the Pthreads implementations with twelve threads using the Euler, RK2 and RK4 integration methods for system size $N = \{512^2, 1024^2, 1536^2...8192^2\}$. The results from the single-threaded CPU implementation are also shown as a reference. The Pthreads implementation provides an average of 6.8x speedup over the single-threaded CPU implementation. The Pthreads multithreading library allows the simulation to make use of all the cores in the CPU.

6.4.2 TBB

For the TBB implementation the number of threads is not explicitly defined, instead the size of the task blocks can be defined. TBB will split the lattice into task blocks of the specified size and distribute them among the available cores. Task blocks of sizes $B = \{2^2, 4^2, 8^2...256^2\}$ have been tested and the results are shown in Figure 6.6. It can be seen from this plot that the performance difference between the different task block sizes does not differ outside the error margin. TBB can make effective use of all the available CPU cores even with very small task block sizes.

There is no significant performance difference between the different task block sizes which makes it hard to choose one size for testing. A task block size of 64^2 has been chosen somewhat arbitrarily as any of the task block sizes in the range tested should provide very similar results. The TBB im-

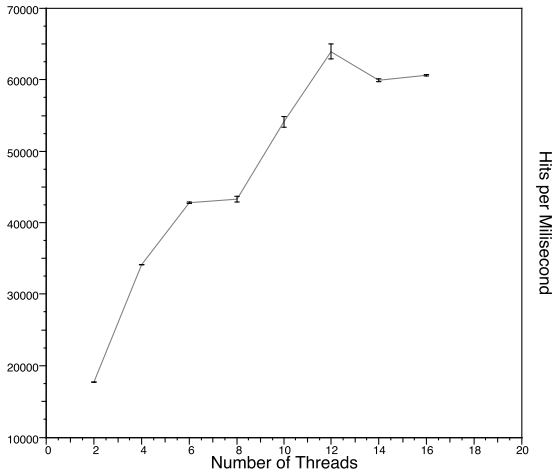


Figure 6.4: Updated lattice points per millisecond vs number of threads for the Pthreads implementation.

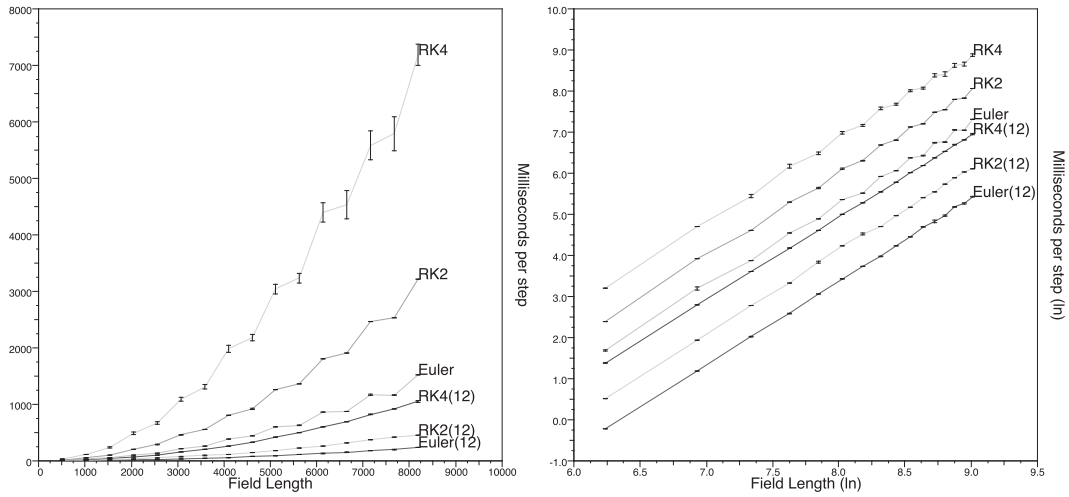


Figure 6.5: Performance comparison of the Pthreads using twelve threads and the single-thread implementations of the Cahn-Hilliard equation using the Euler, RK2 and RK4 integration methods.

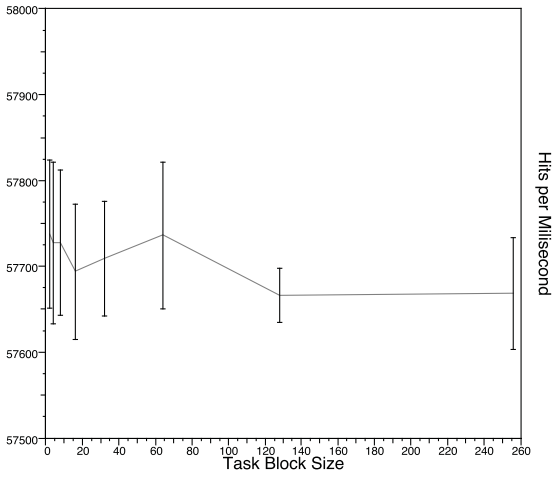


Figure 6.6: Lattice cells updated per millisecond vs TBB task block size.

plementations of the simulations using the Euler, RK2 and RK4 integration methods for system sizes $N = \{512^2, 1024^2, 1536^2 ... 8192^2\}$ have been tested and the results are shown in Figure 6.7.

It can be seen from these plots that the TBB implementation provides a useful speedup for all the tested system sizes. TBB can distribute the task blocks to make use of the CPU cores for all the different system sizes. Once again there are some kinks visible in the performance plots for these simulations, however, they are not as prominent as they are in the single-core implementation. As the multiple cores use both shared and individual cache areas, any caching effects will be spread between these cores.

On average the TBB implementation provides a 6.2x speedup over the single-threaded implementation. This performance improvement is slightly less than the Pthreads implementation on the same hardware which shows that the lower-level and harder to write Pthreads implementation can make slightly better use of the available hardware. However, TBB provides a simple programming interface for implementing the simulations and can still provide comparable performance.

6.5 Cluster

The MPI implementations have been tested on a cluster machine which is comprised of a number of networked workstations. This cluster of workstations or COWS configuration will not represent the best performance for a cluster machine due to the relatively low speed of the network (1Gbps). This workstation cluster contains 18 nodes which each contain an Intel Core 2 Quad Q9400 2.66 GHz with 4GB of DDR2-800 RAM. Each of these nodes are slower than the machine used to test the single CPU simulations but collaboratively can produce more computational throughput.

When computing the MPI implementation discussed in Chapter 5, there will be a tradeoff between computation and communication. As the number of nodes is increased, they can provide

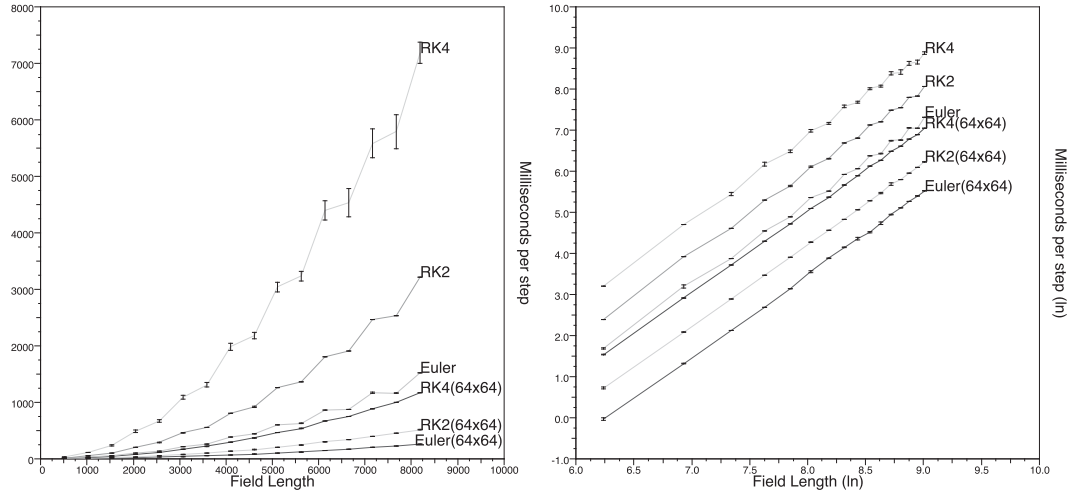


Figure 6.7: Performance of the TBB implementations of the Euler, RK2 and RK4 integration methods as compared to the single-threaded implementation. Results are shown in milliseconds per time step(left) and in ln-ln scale(right).

more overall computational power. However, the amount of communication between nodes will also increase. This will cause a point at which using more nodes will actually increase the time taken to compute the simulation for some system sizes. This effect can be seen in Figure 6.8.

From these performance plots it can be seen that for a system size of $N = 8291^2$, the configuration which uses sixteen nodes provides the best overall performance. For this system size, the communication required by thirty-two nodes outweighs the increase in computational power. It is expected that such configurations will provide improved performance for larger systems which have a higher computation to communication ratio.

The performance of this MPI implementation has been compared to the single-thread implementation for the Euler, RK2 and RK4 integration methods. This comparison is shown in Figure 6.9. From these figures it can be seen how variable the performance of such a cluster can be. Because of the uncontrollable network traffic on such a cluster, the communication times can be very variable. However, from this plot the scaling of this MPI implementation can still be seen.

It can be expected that different lattice decomposition methods will have an effect on performance. The implementation tested here uses a one-dimensional lattice decomposition method which will does not provide a good computation to communication ratio. A higher-dimensional method will require less communication between nodes and will thus scale better to a large number of nodes.

6.6 Tesla GPU

The GPU algorithms discussed in Chapter 5 have been implemented and optimised for Tesla architecture GPUs. These simulations have been tested on an NVIDIA GTX260 with 896MB of GDDR3

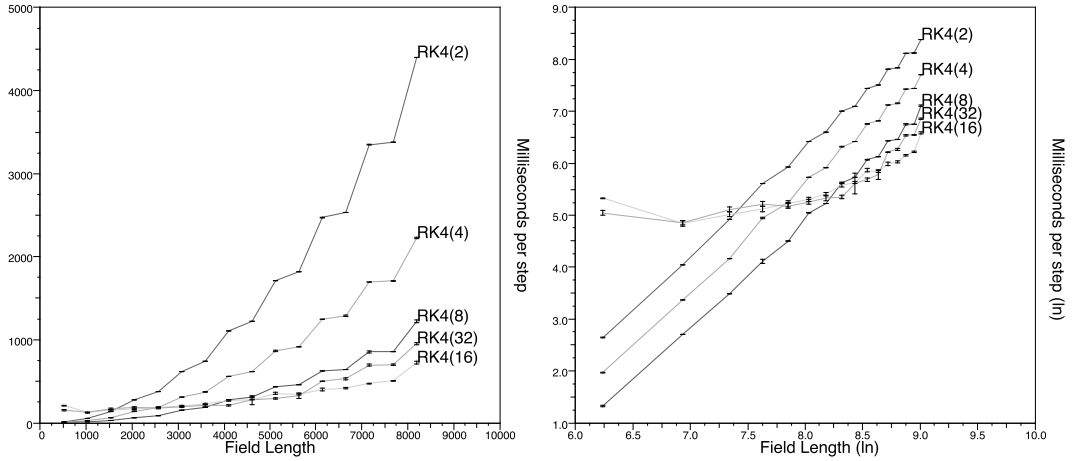


Figure 6.8: Performance comparison of MPI nodes computing a finite-differencing simulation using the RK4 integration method, tested with $\{2, 4, 8, 16, 32\}$ nodes.

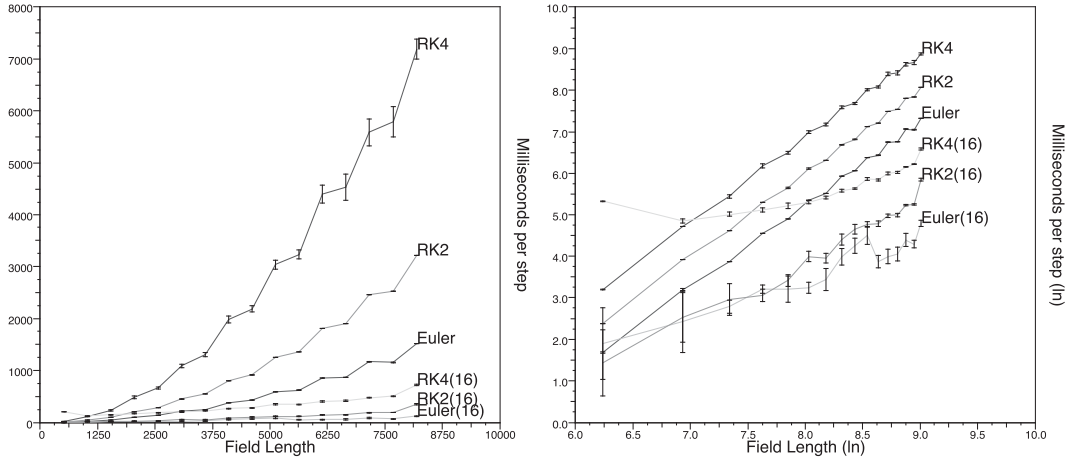


Figure 6.9: Performance comparison of MPI implementation and the single-thread simulations. Implementations of the Euler, RK2 and RK4 methods have been tested for systems of size $\{512...8291\}$. Results are shown in milliseconds per time step (left) and ln-ln scale (right).

memory and 216 stream processors. Three different implementations have been tested on this graphics card, each of which use different GPU memory types - global, shared and texture. These three implementations are discussed in depth in Chapter 5.

Each of these implementations have one updating thread per lattice cell. The main method of configuring these simulations is to control the size of the thread block. Each Tesla thread block has a width of 16 threads to ensure coalesced global memory access whenever possible. The performance of the implementations has been tested with a height of $B = \{1, 2, 4...256\}$ and plotted in Figure 6.10.

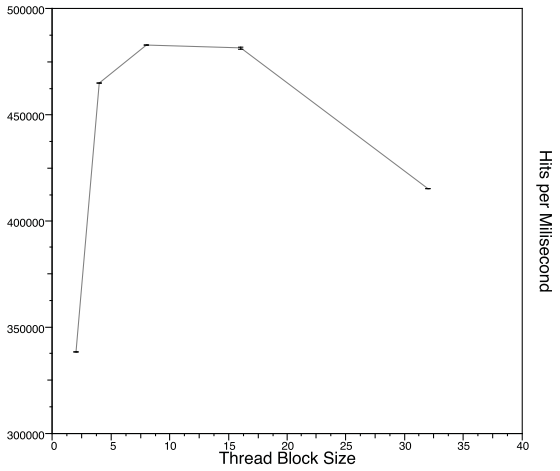


Figure 6.10: The performance of different size thread blocks for the Tesla architecture GPU implementation.

From this plot it can be seen that the optimal thread block size is 8x16. This thread block size has been used to test and compare the three different Tesla kernel implementations. The performance plots for these implementations using the Euler, RK2 and RK4 integration methods and system sizes $N = \{512^2, 1024^2, 1536^2...8192^2\}$ are shown in Figure 6.11. The simulations using the RK4 integration method require the most memory storage. This limits the maximum system size that can be computed on the GTX260 graphics card to 5632^2.

From this plot it can be seen that the kernel using texture memory provides the best performance. The shared memory kernel does provide some performance benefit over the simple global implementation but the spatial cache of the texture memory still provides the best performance on the GTX260. These results confirm the findings published in [102] which tested these implementations on an 8800GTX.

The performance of the texture memory kernel implementation has been compared to the single-thread CPU simulation. The results of these simulations using the Euler, RK2 and RK4 methods are shown in Figure 6.12. These plots show that the GTX260 can provide many times the performance of a CPU core. A GTX260 graphics card can provide an average speedup of 55x over one core of an Intel i7-970.

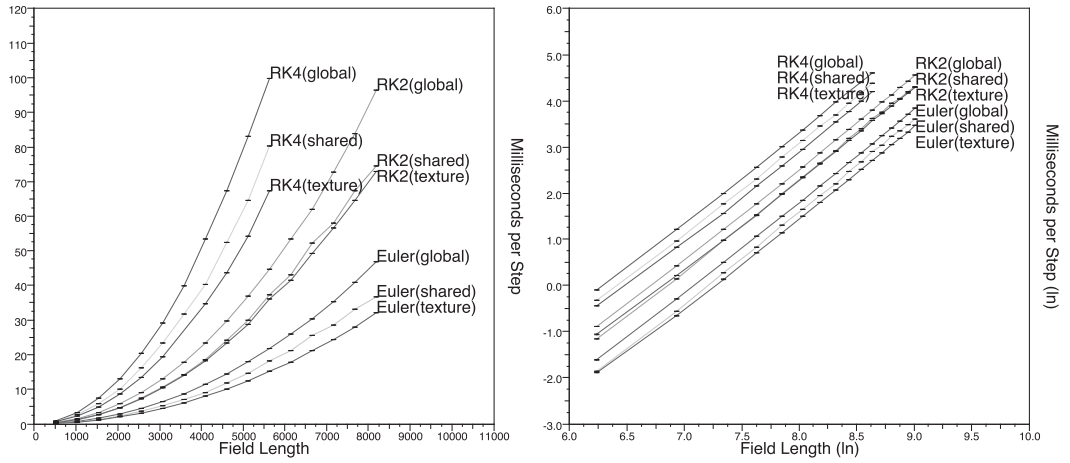


Figure 6.11: Comparison of GPU memory types for simulations executing on a Tesla architecture GPU. Results are shown for the Euler, RK2 and RK4 integration method.

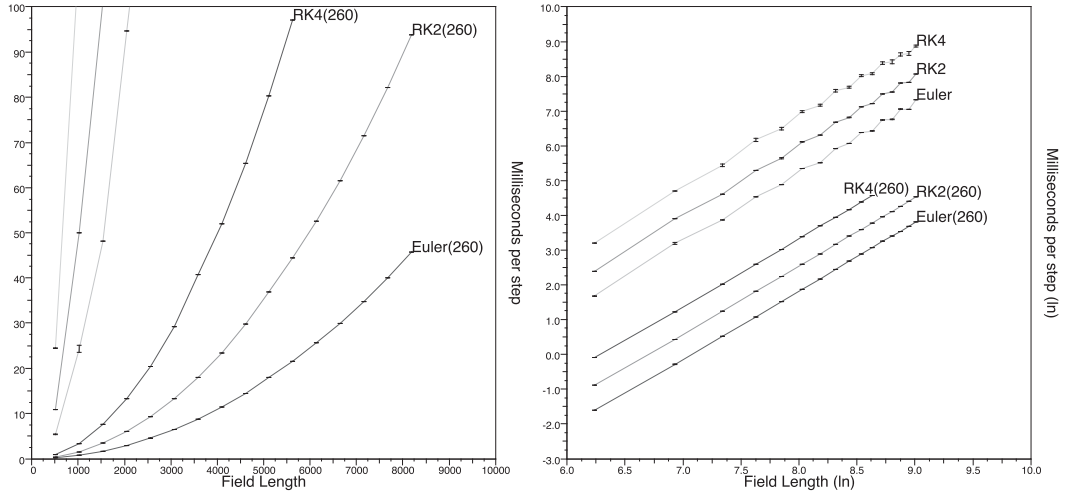


Figure 6.12: Comparison of the GPU Tesla implementation and the single-threaded C simulations.

6.7 Fermi GPU

To test the the Fermi architecture GPUs an NVIDIA GTX480 card has been used. This is an NVIDIA computing processor with 480 cores and 1.5GB of GDDR5 device memory. It is one of the GeForce series graphics cards designed for computer games. The performance of this card is compared to a GeForce GTX580 and the high-performance computing C2070 card in Section 6.10.

First the optimal size of thread block is determined for the simulation. For the Fermi architecture the thread blocks are of width 32 to ensure coalesced global memory access. Thread blocks of size $B = \{32, 64, 128...1024\}$ have been tested and the results shown in Figure 6.13. This plot shows the optimal thread block size is 8x32.

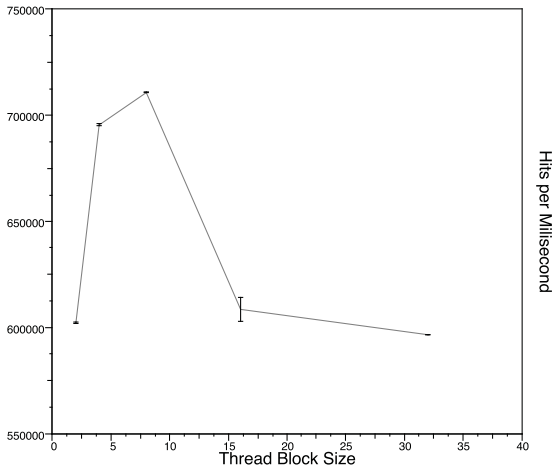


Figure 6.13: Performance of different size thread blocks for Fermi architecture GPUs.

The Fermi GPUs have been tested with the same simulations using three different types of GPU memory - global, shared and texture. These implementations have been tested with the Euler, RK2 and RK4 integration methods and the results shown in Figure 6.14.

From this figure it can be seen that the global and shared memory implementations offer very similar performance. For the Euler and RK2 methods the shared memory implementation offers slightly better performance. For the RK4 method the global memory offers a performance increase. For the Fermi architecture the use of texture memory reduces the performance of the simulations.

Finally the Fermi global memory implementation is compared to the single-core CPU simulation. The Fermi GPUs offer a major performance gain over the CPU as can be seen in Figure 6.15. From this plot it can be seen that the GTX480 offers a speedup of approximately 85x over the CPU simulations.

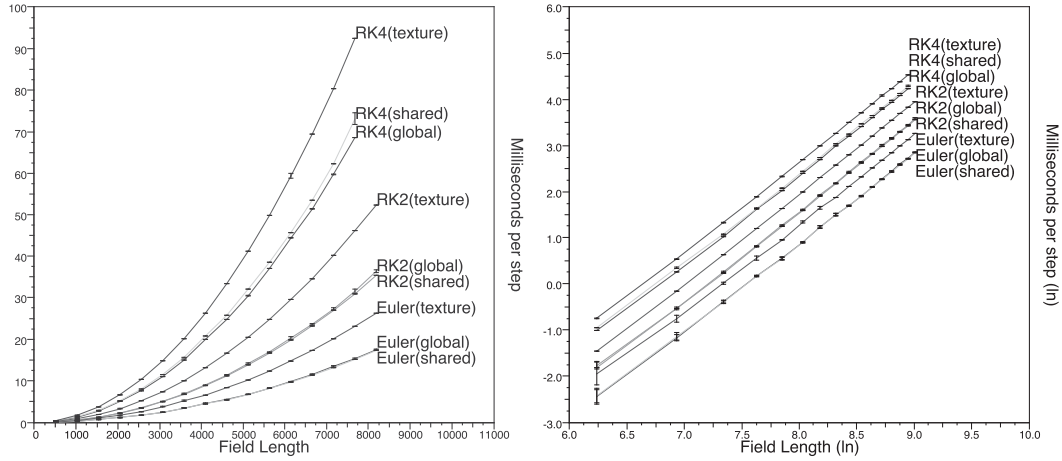


Figure 6.14: Comparison of the Fermi simulations using global, shared and texture memory. Plots are milliseconds per time step vs field length (left) and ln-ln scale (right).

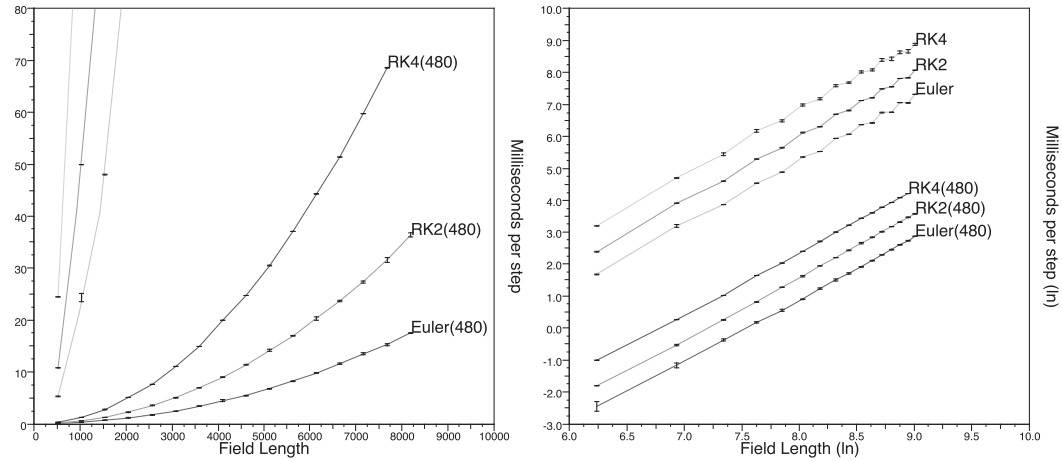


Figure 6.15: Performance comparison of the CUDA global implementation and the single-threaded C implementation.

6.8 Multi-GPU

The multi-GPU implementation used to test the performance of a multi-GPU architecture is the implementation which uses the asynchronous border communication method. This method is presented in Chapter 5 along with the synchronous border communication. It has been previously shown that the asynchronous communication method performs better than the synchronous method [125, 126].

This method has been implemented with kernels that use global memory to access the lattices. The implementation has been tested using two and four NVIDIA GTX480 graphics cards and compared to the global implementation on a single GTX480 and the single-core CPU simulation. The results of this comparison are shown in Figure 6.16.

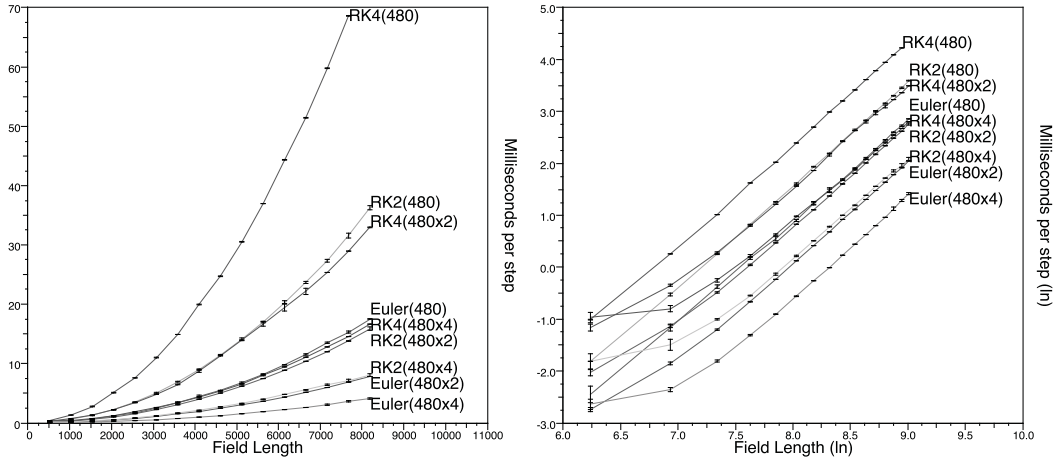


Figure 6.16: Performance of the multi-GPU implementation on two and four GeForce GTX 480 graphics cards. These results are compared to the single-GPU implementation on a single GTX480.

Note the curve in the performance plots of the multi-GPU simulations. For small system sizes the simulations cannot make full use of the computational power of the GPU devices. The asynchronous border communication method allows the use of multiple GPU devices with an almost linear speedup for system sizes $N > 2048$. This agrees with the results published in [125, 126] which investigated the multi-GPU implementations using only the Euler integration method.

6.9 GPU Cluster

Unfortunately no results have been collected for the GPU-accelerated cluster implementation. The only GPU-accelerated cluster at Massey University is a cluster of workstations in everyday use. With other users and applications executing on this cluster it is not possible to collect consistent performance data. The simulations executed on this cluster showed highly variable results varying by

many orders of magnitude depending on the specific nodes used, other applications executing and the network traffic. This cluster does not fairly represent the performance of a cluster with GPU-accelerated nodes. Results of the decoupled communication implementation published in [135] show that finite-difference applications can scale well on GPU-accelerated clusters.

6.10 Performance Comparison

Finally the performance of all the different parallel architectures have been compared to show relative performance. Various models of each architecture have been tested with the optimal implementation of the Cahn-Hilliard simulation using the RK4 integration method. The performance plots comparing these different architectures are shown in Figure 6.17.

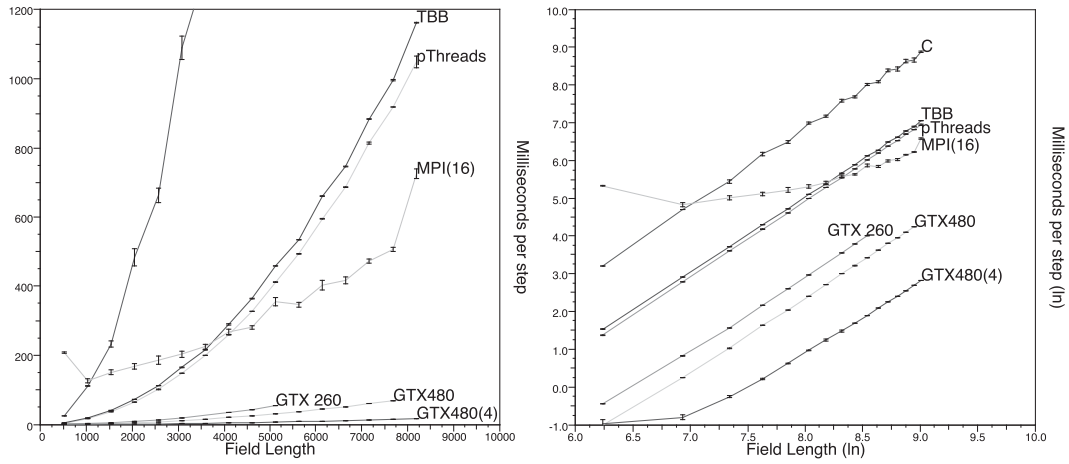


Figure 6.17: Performance results of the different parallel computing architectures. Plots show the time taken to compute one step of a Cahn-Hilliard simulation using the RK4 integration method.

Table 6.1 shows a performance comparison of various architectures and models. The table shows the architecture family, specific model, implementation used and the total memory and speedup each model provides. The speedup of each architecture has been calculated by comparing the time taken to compute this simulation for a system size of 8192^2 (or the largest system the architecture is capable of computing) and compared to the single-core CPU implementation executed on an Intel i7-970. The maximum system size each architecture can support will depend on the integration method used and the storage required for each system state. Instead of listing the maximum system size each architecture can support, the type and total memory of each architecture is listed.

It is clear from these collected results in Figure 6.17 and Table 6.1 that the GPU-based processing architectures can provide the highest overall performance. Correct use of a multi-threading library such as Pthreads or TBB can allow an implementation to make effective use of the multiple cores available on modern processors. The MPI implementation shows reasonable speedup across

Table 6.1: Performance comparison of different architectures computing the Cahn-Hilliard simulation using the RK4 integration method.

Architecture	Model	Implementation	Total Memory	Speedup vs CPU
CPU	Intel 970	Single-thread C	DDR3 12GB	1.0x
CPU	Intel 970	Multi-thread Pthreads	DDR3 12GB	6.8x
CPU	Intel 970	Multi-thread TBB	DDR3 12GB	6.2x
CPU-Cluster	Intel Q9400(2)	MPI	DDR2 8GB	1.6x
CPU-Cluster	Intel Q9400(4)	MPI	DDR2 16GB	3.2x
CPU-Cluster	Intel Q9400(8)	MPI	DDR2 32GB	5.9x
CPU-Cluster	Intel Q9400(16)	MPI	DDR2 64GB	9.9x
Telsa GPU	GTX260	CUDA (texture)	GDDR3 896MB	55x
Fermi GPU	GTX480	CUDA (global)	GDDR5 1.5GB	85x
Fermi GPU	GTX580	CUDA (global)	GDDR5 1.5GB	120x
Fermi GPU	C2070	CUDA (global)	GDDR5 6GB	75x
Multi-GPU	GTX480 (2)	CUDA (global)	GDDR5 3GB	220x
Multi-GPU	GTX480 (4)	CUDA (global)	GDDR5 6GB	430x
Multi-GPU	C2070(2)	CUDA & Pthreads	GDDR5 12GB	165x

distributed nodes in a cluster of workstations but the performance is limited by the quality of the interconnect. Finally the GPU and multi-GPU implementations show very impressive performance providing 55-430x speedup over one core of an Intel 970X. The multi-GPU implementations often show superlinear speedup over a single GPU. It is believed this is due to the devices being able to make better use of the L1/L2 caches. It is clear that these GPU architectures are very well suited to processing the regular finite-difference simulations.

6.11 Conclusions

The results presented in this chapter show that parallel computing can significantly reduce the time taken to compute these simulations. The GPU implementations offer the best performance compared to the other parallel processing architectures. These implementations are harder to program and more sensitive to configuration but can offer high computational throughput for low cost. These performance results show that the ability to produce parallel code for a variety of parallel architectures is an important feature of any simulation generator. Without the ability to produce code for these different architectures, simulations will always be severely limited in the size of system and simulation length that can be computed.

"Words offer the means to meaning and for those who will listen, the enunciation of truth."

Hugo Weaving in James McTeigue's *V for Vendetta*

7 Simulation Description Language

7.1 Introduction

Generative Programming (GP) systems automatically produce software solutions from high-level descriptions of the problem [42]. These GP systems are conceptually very similar to an industrial assembly line. Each system produces software in the same program family, but each solution can be configured with different combinations of components. Creating a GP system requires a program family to be identified, the common features of this program family to be determined and a Domain-Specific Language (DSL) to be developed to allow specific solutions to be defined [43, 44].

The DSL should be capable of describing the full range of solutions within the problem domain. From this description, the GP system will assemble the required software components to produce the target implementation. The GP assembly line automatically maps definitions from problem space to solution space implementations [42].

The program family identified and described in this thesis is parallel simulations for partial differential field equations. A generative programming system that produces these simulations must be able to correctly assemble software components for a wide range of possible simulations. To make this goal achievable, the problem domain is restricted to a specific type of simulation which allows some requirements for the system to be determined.

This GP system should be capable of producing simulations for partial differential field equations of the type described in Chapter 2. These are field equations which can be solved using rectilinear lattices, finite-difference methods and explicit Runge-Kutta integration methods. Simulations of these equations can be constructed in any number of dimensions. In order to do this the system must be able to manipulate finite-differencing stencils and substitute them into the equations. There is also a large family of explicit Runge-Kutta methods which can be used to integrate the equations over time. The system must be able to correctly apply these integration methods to the equations. Finally the system must be capable of generating parallel code to compute these simulations without enforcing any restrictions on language, architecture, precision or method of parallelisation.

A GP system which can produce such simulations would automate the time-consuming task of

producing these simulation codes by hand. Many simulation codes are similar, but hand-crafting them can often be error-prone and a large amount of development time is spent debugging. Another advantage of such a system is the ease with which simulations can be migrated to new parallel computing architectures. Rather than re-writing each simulation using the new parallel language/library, a single new mapping can be created which can construct simulations using this new language/library. This is especially useful for research groups with a large simulation base as the entire simulation collection can be migrated to the latest parallel computing architecture with the creation of a single mapping.

Creating a GP assembly line requires the development of two major components - a Domain Specific Language to describe the simulations in an abstract way and a method of mapping this abstract description to a target solution. This chapter describes the DSL developed to describe the simulations, Chapter 8 discusses how these simulations are internally represented and manipulated and finally Chapter 9 discusses how these simulation representations are mapped to target implementations. The GP system described in this thesis is named *Simulation Targeted Automatic Reconfigurable Generator of Abstract Tree Equations* or STARGATES.

7.2 Domain Specific Language

Automatically generating parallel solvers for simulations requires the development of a DSL to describe them. This DSL must be capable of describing all the different parts of the simulation - equation, stencils, boundary conditions, integration method and simulation configuration. It is important that the definition and configuration of the simulation should not enforce any limitations on the target implementations.

There are five main features of this type of simulation that must be defined by the high-level description - equation, stencils, integration method and configuration. These five features of the simulation are very different in nature. The field equation is defined by one or more mathematical expressions, the stencils are discrete operators, the integration methods are a set of mathematical expressions and configuration is a selection of options for the simulation. Because these features are different in nature and are logically distinct it was decided that they should be defined separately in a way that is appropriate to that feature. Equations, stencils, boundaries and integration methods are all defined in separate ways and selected according to the configuration.

The four different elements and the methods of describing them are discussed in Sections 7.3, 7.4, 7.5 and 7.6.

These elements define all the necessary information about a simulation but do not define an implementation. Many different implementations, either parallel or sequential, can be constructed to compute the same simulation. For this reason all details of parallel implementation are decided by the target generator and not by the simulation DSL. However, the configuration can contain information about how best to parallelise the simulation which can then be used by the generators. The details of these target generators are discussed in Chapter 9.

7.3 Equation Parser

The field equations that govern the behaviour of the system are one or more equations containing standard mathematical expressions and discrete operator stencils. The easiest way for a user to define such an equation is to write it as text in a mathematical form. The equation parser reads equations written in ASCII in a mathematical form. This parser can read the equation(s) and construct an internal representation which can be manipulated and combined with other elements of the simulation. The parser assumes that the equation(s) given can be simulated using finite-difference methods and explicit Runge-Kutta integration. It is still the responsibility of the user to determine whether equations can be correctly simulated using these methods.

While these equations almost always contain some kind of discrete stencil, these are defined separately and referred to by the equation definition. Many different equations will use the same stencils and it does not make sense to force the user to redefine them in every equation that uses them. The three equations used as examples in this thesis all use the Laplace operator and the Cahn-Hillard equation also uses the Biharmonic operator. The method for defining these stencils is discussed in Section 7.4.

In addition to the equations that govern the system, this part of the simulation description also defines the lattices and adjustable parameters of the system. This is the definition of the field which the simulation approximates and the conditions of the system such as heat. It is important to define which parameters of the equation are the lattices that represent that system and which are the controllable parameters. These are listed as either `<type> <name>` for the parameters or `<type>[] <name>` for lattices.

This equation parser has been implemented using the modern compiler generator technology ANTLR [147]. ANTLR is a relatively modern tool which builds upon historical developments [148] including the well-known lexing/parsing tools: lex/yacc [149] and flex/bison [150, 151]. ANTLR allows a relatively simple grammar to be specified from which ANTLR will automatically generate a Lexer and a Parser. The grammar shown here supports the declaration of parameters, lattices and equations with simple mathematical operators $+, -, *, /$ as well as some common mathematical functions. This grammar is shown in Listing 7.1.

Listing 7.1: Simple equation ANTLR grammar.

```
fragment DIGIT : '0'..'9';
fragment CHAR  : 'a'..'z'|'A'..'Z'|'_';

FUNC   : ('ABS' | 'SQRT' | 'EXP');
ID     :  CHAR (CHAR|DIGIT)*;
NUM    : (DIGIT)+ ('.' (DIGIT)+ )?;
LPAREN : '('  ;
RPAREN : ')'  ;
STAR   : '*'  ;
SLASH  : '/'  ;
CARET  : '^'  ;
PLUS   : '+'  ;
MINUS  : '-'  ;
SEMI   : ';'  ;
EQUAL  : '='  ;
START  : 'd/dt';
```

```
LSQUARE : '['   ;
RSQUARE : ']'   ;
LCURLY  : '{'   ;
RCURLY  : '}'   ;

file                    : (statement)+ EOF!;
statement               : (declaration|equation);
declaration             : type=ID LSQUARE RSQUARE name=ID SEMI
                        | type=ID name=ID SEMI
                        ;
equation                : START ID EQUAL additiveExpression SEMI
additiveExpression      : multiplicativeExpression ((PLUS^|MINUS^) multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR^|SLASH^) unaryExpression)*;
unaryExpression         : atom
                        | MINUS atom
                        ;
atom                    : NUM
                        | FUNC LCURLY additiveExpression RCURLY
                        | ID
                        | LPAREN additiveExpression RPAREN
                        | name=ID LCURLY additiveExpression RCURLY
                        ;
```

Parsing mathematical equations is potentially an open-ended problem. The parser is limited to reading field equations in relatively simple form using common operations. The advantage of using an ANTLR-based equation parser is that it is very easy to modify the parser and incorporate new features into it.

The Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations discussed in Chapter 2 are used as examples to show how equations can be represented in this ASCII form. These examples are presented in Sections 7.3.1, 7.3.2 and 7.3.3.

7.3.1 Parsing the Cahn-Hilliard Equation

The Cahn-Hilliard equation discussed in Chapter 2 models the phase separation of a binary alloy. This equation has a single scalar field u which represents the ration of A- over B-atoms and four adjustable parameters M, B, U and K. The equation is given again in equation 7.1 for ease of reference.

$$\frac{\partial u}{\partial t} = M\nabla^2\left(-Bu + Uu^3 - K\nabla^2 u\right) \tag{7.1}$$

This equation can be written in an ASCII form that can be read by the equation parser. The description defines that there are four adjustable parameters `M`, `B`, `U`, `K` and a lattice `u` which are all of type `floating`. The time derivative of the lattice `u` is given as a mathematical expression involving the lattice, the parameters and the discrete operator `Laplacian`. The ASCII representation of the Cahn-Hilliard equation can be written as:

```
floating M;
floating B;
floating U;
floating K;
floating[] u;
d/dt u = M * Laplacian{(-B*u + U*(u*u*u) - K*Laplacian{u})};
```

It is important at this point to specify the meaning of the type `floating` in this representation. This does not necessarily mean that any code generated will use 32-bit floating point variables for these parameters and lattice. Different programming languages will use different names for floating point values with different levels of precision. The design philosophy of STARGATES is to leave decisions on all implementation details to the code generators. Thus the type is defined here as `floating` but the output generators may use `float`, `double`, `REAL` or any other suitable floating point data type. This way the equation description does restrict the precision of the final code in any way.

7.3.2 Parsing the Ginzburg-Landau Equation

The Ginzburg-Landau equation provides a useful model of the thermodynamic and macroscopic properties of superconductors. In this equation there are three parameters `P`, `Q`, `i` of type `complex` and one parameter `Y` of type `floating`. The parameter `i` is not really a parameter of the model but actually represents the complex number $0 + 1i$. The system this model approximates can be represented by a single lattice of complex numbers `u`. The equation from Chapter 2 is re-stated in equation 7.2.

$$\frac{\partial u}{\partial t} = -\frac{p}{i}\nabla^2 u - \frac{q}{i}\left|u\right|^2 u + \gamma u \tag{7.2}$$

The equation can be represented in ASCII as:

```
complex P;
complex Q;
complex i;
floating Y;
complex[] u;
d/dt u = -P/i*Laplacian{u} - Q/i*ABS(u)*ABS(u)*u + y*u
```

The interesting point of difference between this equation and the Cahn-Hilliard equation is the data type of the model. The Ginzburg-Landau equation deals with a complex field, thus the data type of the lattice is a complex number. Once again this complex number type specifies a type which must be interpreted by the target code generators. Different languages will have different data types and methods available which will depend on the architecture and language/libraries available. Thus the details of how the data type is actually implemented must be made by the generator.

In Chapter 2 it was shown how the Ginzburg-Landau equation can be split into real and imaginary parts. This results in two lattices which represent the different parts of the complex field and two equations to govern the behaviour of the system. This is a different method of computing the same model and can be represented in ASCII as:

```
floating Pi;
floating Pr;
floating Qi;
floating Qr;
```

```
floating Y;
floating[] ui;
floating[] ur;

d/dt ui =   Pr*Laplacian{ur}         - Pi*Laplacian{ui}         +
            Qr*(ur*ur*ur - ui*ui*ui) - Qi*(ur*ui*ui + ur*ur*ui) +
            Y *ui;
d/dt ur = - Pi*Laplacian{ur}         - Pr*Laplacian{ui}         -
            Qi*(ur*ur*ur - ui*ui*ui) - Qr*(ui*ui*ur + ui*ur*ur) +
            Y *ur;
```

7.3.3 Parsing the Lotka-Volterra Equation

The Lotka-Volterra equations describe the interactions of two or more species populations. The equations for a two species predator-prey system using the Laplacian operator for spatial coupling is given in equation 7.3. This is a restatement of the equations in Chapter 2. This model has two floating point type lattices which represent the populations of each species. The equation can be controlled by six floating point parameters $A, B, C, D, D0$ and $D1$. This model has two governing equations which calculate the derivatives of the two populations

$$\begin{aligned} \frac{du_0}{dt} &= D_0\nabla^2 u_0 + Au_0 - Bu_0u_1 \\ \frac{du_1}{dt} &= D_1\nabla^2 u_1 + Du_0u_1 - Cu_1 \end{aligned} \tag{7.3}$$

This predator-prey model can be written in ASCII as:

```
floating A;
floating B;
floating C;
floating D;
floating D0;
floating D1;
floating[] u0;
floating[] u1;
d/dt u0 = D0*Laplacian(u0) + A*u0 - B*u0*u1;
d/dt u1 = D1*Laplacian(u1) + D*u0*u1 - C*u1;
```

Note that in this model there are two lattices `u0` and `u1` and two governing equations `d(u0)/dt` and `d(u1)/dt`. This is an example of how multiple interacting fields and coupled equations can be represented in ASCII form. This example has two coupled equations and lattices but the equation parser is capable of dealing with models with any number of equations.

7.4 Stencil Operators

To produce simulation code, STARGATES must be capable of reading and manipulating the finite-differencing stencils discussed in Chapter 3. One advantage of using a generator is that the computation model can be specified independently of whether it is to be solved on a two-, three- or higher-dimensional lattice. Many PDE problems that arise in areas of physics can be described in higher dimensions and it is useful to be able to separate the dimensionality from other model details. This makes it possible to generate simulations for an arbitrary number of dimensions.

To generate simulation code, the stencil library must be capable of providing stencils of any dimensionality. This allows the equations to be defined regardless of the number of dimensions. When a simulation of a specific dimensionality is generated, the stencil library will construct the required stencils depending on the equation and the specified dimensionality of the simulation. To support this hyper-dimensional functionality, the STARGATES code generator makes use of the hypercube apparatus described in [152].

These stencils often have floating-point or integer data types but are not limited to this type. Stencils of any data type can be constructed and manipulated for use within the GP system [153]. The storage and manipulation of the stencils should not enforce any limitation on the precision of the stencil coefficients. For this reason the stencils are parsed as an expression which can later be evaluated to a specific value. This way any decision about the precision of the stencil coefficients is decided by the target generator and not within the system.

7.4.1 Defining Stencils

STARGATES provides three pre-defined stencils which can be used by equations. These stencils are the Laplacian, Biharmonic and SpatialAverage. These stencils are generated within the STARGATES system which can construct the stencils for any number of dimensions. The three stencils in two- and three-dimensions are shown in Figure 7.1.

While STARGATES can generate these stencils, it also allows users to define their own stencils with whatever size and values are desired. These stencils can be constructed with any data type and can be manipulated for use within the simulations [153]. To define a stencil, four attributes must be given - the data-type of the stencil, the number of dimensions, the size of the stencil in each dimension and the actual coefficients of the stencil cells. The following structure is used to define these attributes.

```
type = `_'
dim = [_,_,_...]
data = [_,_,_,_...]
```

The item `type` obviously defines the data type of the stencil. This can be any data type the user desires but it must obviously be supported by the target language for the final simulation. `dim` is an array of integers that controls both the number of dimensions and the size of the stencil in each. The length of the array defines the number of dimensions while the values of the array define the size of

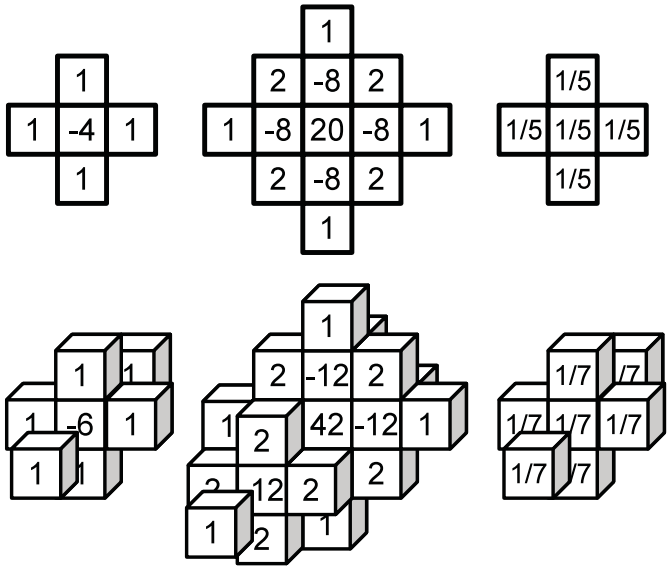


Figure 7.1: The three stencils generated from within STARGATES in two- and three-dimensions. The stencils are Laplacian (left), Biharmonic (centre) and SpatialAverage (right).

the stencil in each dimension. Finally the `data` array contains the actual values of the stencil. Using this structure the Laplacian operator can be defined in two dimensions as:

```
type = 'integer'
dim = [3,3]
data = [0,1,0,1,-4,1,0,1,0]
```

To allow the user to define a stencil for different numbers of dimensions, STARGATES allows multiple stencils with the same name to be defined as long as they differ in the number of dimensions. This way users can define custom stencils for any number of dimensions to allow simulations to be generated with any dimensionality. When a simulation is generated, the system will select the stencil with the appropriate number of dimensions and substitute it into the equation.

There is one special case where this is not necessary. If a stencil (such as the Laplace operator) can be constructed by overlaying the one-dimensional stencil over itself in each dimension, STARGATES can automatically construct these higher-dimensional stencils. To construct a higher-dimensional stencil, the one-dimensional stencil is rotated around the central point and overlaid onto the original stencil. The overlapping stencil cells are then added together to give the new stencil values. This process can be seen in Figure 7.2.

This process of automatically constructing high-dimensionality stencils from the one-dimensional stencil may not always be valid. It is the responsibility of the developer to use this feature wisely. If a stencil cannot be extended to a higher dimension automatically, the correct stencils in each dimension should be explicitly defined.

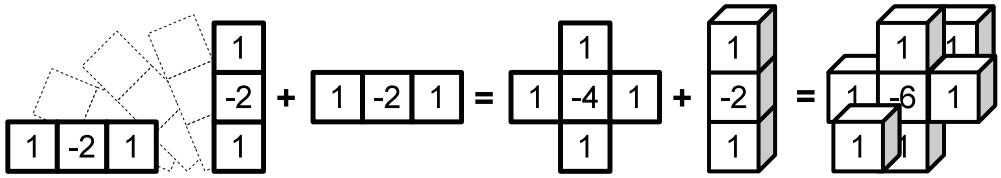


Figure 7.2: The one-dimensional Laplace stencil being rotated and overlaid on itself to construct the higher dimensional stencils.

7.5 Integration Methods

To compute how a simulation evolves over time, a suitable integration method must be used. The family of simulations currently targeted by STARGATES are restricted to using explicit Runge-Kutta integration methods. The details of these integration methods are fully discussed in Chapter 3. The advantage of these methods is that they are easy to parallelise as each time step can be explicitly computed without the need for iteratively solving a solution.

These integration methods are a series of mathematical expressions which define the computation of intermediate stages and the final computation to give the a solution at the end of the time step. One method of defining the integration methods would be to allow them to be written in ASCII as a series of mathematical expressions. These expressions could be read using the same technology used to create the equation parser.

However, Chapter 3 offers an alternative way to easily define explicit Runge-Kutta integration methods. Butcher tableaux can be used to represent explicit (and implicit) Runge-Kutta integration methods [62]. Each tableau is a table of coefficients which shows how to combine function evaluations at each stage to compute the next. The general form of a Butcher tableau which defines an explicit integration method is given in Table 7.1.

Table 7.1: General form of a tableau representing an explicit integration method.

0					
c_1	$a_{1,0}$				
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_{s-1}	$a_{s-1,0}$	$a_{s-1,1}$	$\cdots$	$a_{s-1,s-2}$	
	b_0	b_1	$\cdots$	b_{s-2}	b_{s-1}

The series of mathematical expressions which performs the actual integration can be constructed from the coefficients in this tableau. As described in Chapter 3 the mathematical expressions to perform the integration can be constructed using the following equations:

$$Y_i = Y_0 + h \times \sum_{j=0}^{i-1} a_j F_j \tag{7.4}$$

$$Y_s = Y_0 + h \times \sum_{j=0}^{s-1} b_j F_j \tag{7.5}$$

where:
Y_0 is the initial system state
Y_s is the final system state
F_i is the evaluation of the model for the system state Y_i

STARGATES can parse a Butcher tableau from a simple ASCII representation and construct the mathematical expressions of the integration method. The values within the tableau are parsed as mathematical expressions rather than fixed values. The advantage of parsing the coefficients of the tableau as equations rather than fixed values is that it allows the decision of implementation and precision to be decided by the generator rather than during the construction of the method. This is in keeping with the STARGATES philosophy of keeping the simulation description as abstract as possible until the generation phase.

The ASCII form for the Butcher tableau is simply the a, b and c coefficients separated by "|" characters and new lines. The general form of this ASCII representation of a Butcher tableau is:

```
0    |         |         | ... |           |
c1   | a1,0    |         | ... |           |
c2   | a2,0    | a2,1    | ... |           |
...  | ...     | ...     | ... | ...       |
cs-1 | as-1,0 | as-1,1 | ... | as-1,s-2 |
b    | b0      | b1      | ... | bs-2      | bs-1
```

The left-most column contains the values of c. While the c values are not necessary for constructing an integration method, they are useful for checking the values of a. The sum of the values of the a coefficients on each row should equate to the value of c on that row. This allows STARGATES to perform some verification of the integration method. The bottom row of the tableau contains the values of b. These values are used for the final computation of solution of the integration method. Finally all other values in the ASCII representation are the values of the coefficients of the method a.

This ASCII form of a Butcher tableau can be used to define any of the integration methods discussed in Chapter 3. For example the tableau for the RK4 integration method can be written in this ASCII form as:

```
0   | 0   | 0   | 0   | 0
1/2 | 1/2 | 0   | 0   | 0
1/2 | 0   | 1/2 | 0   | 0
1   | 0   | 0   | 1   | 0
```

```
b   | 1/6 | 1/3 | 1/3 | 1/6
```

This method of defining integration methods allows the user to define potential complex integration methods in an easy to read and understand way. These integration methods definitions are currently limited to explicit Runge-Kutta methods but could also be used to define implicit Runge-Kutta methods.

These integration definitions describe the method but do not provide any implementation details. This is intended for defining the simulation and not the implementation. The final target code generators must determine how to compute the simulation using these methods. They must consider factors such as parallelisation, memory usage and communication. These target generators are discussed in Chapter 9.

7.6 Configuration

The final specification of the simulation is the configuration information. The configuration is where the user actually defines what solution STARGATES should assemble. The equation, stencils and integration methods are intended to be rarely modified as they should remain the same for any simulation implementations produced. The configuration simply selects a set of options which controls what the target generator produces and is the highest-level description of the simulation.

This configuration format is simply a set of key-value pairs that represents the desired simulation options. There are some important keys which define the required information of the simulation. The required keys are **equation**, **integration**, **dimensions**, **boundary** and **target**. These are necessary configuration options which are required for STARGATES to assemble a solution. These key value pairs are provided in the following format:

```
<equation>    | <value>
<integration> | <value>
<dimension>   | <value>
<boundary>    | <value>
<target>      | <value>
<key>         | <value>
<key>         | <value>
...
```

In addition to these required configuration values, additional key-value pairs may also be defined. These optional configuration values may or may not be used by the target generators. These key-value pairs can represent a range of different options which can be target-specific or general options: preferred decomposition method, specific optimisations etcetera.

Each target generator will configure the solutions they produce depending on these key values. It is the responsibility of the developer to manage these keys and write the target generators to make best use of the information provided. These target code generators are discussed in Chapter 9.

7.7 Conclusions

This chapter has discussed how a Domain Specific Language can be constructed to define the necessary elements of a field equation simulation using finite-differencing and explicit Runge-Kutta integration methods. Each of these elements is different in nature and may be combined in various ways. For this reason the equations, stencils and integration methods are defined separately and selected by the simulation configuration.

The equations can be written in ASCII in a mathematical form which is then read by the equation parser. This allows the user to write the equations in a similar representation to their mathematical form. The stencil operators can be defined in ASCII for multiple dimensions or constructed from a single one-dimensional stencil. The integration methods can be defined as a Butcher tableau from which the integration method can be constructed. Finally the simulation configuration is defined using a set of required and optional key-value pairs.

"Any intelligent fool can make things bigger and more complex. It takes a touch of genius - and a lot of courage to move in the opposite direction."

Albert Einstein

8

Simulation Representation

8.1 Introduction

Chapter 7 has discussed a Domain Specific Language which has been developed to describe the required elements of finite-differencing simulations. This chapter describes how these definitions are internally stored and manipulated to represent all the information required to define a simulation. It is this internal representation that is interpreted by the target code generators described in Chapter 9 to generate the target source code.

The equations, stencils and integration methods are defined separately and the configuration information selects which of these should be combined together to form a specific simulation. At this point the simulation information is still separate from any implementation or target-specific details. Combining these separate elements requires them to be represented in a manner that can be manipulated and combined together. These elements are read from their DSL definition and converted to trees. Representing these different components as trees allows them to be easily manipulated and combined into a single tree that represents the entire simulation. This tree can then be traversed by the code generators to produce the final target simulations.

The process of internally representing the simulation parts as trees and combining them together is the first part of the Generative Programming process that maps the simulation from the problem space to the solution space [42]. The different features can be combined together regardless of the target architecture for a specific implementation. The second part of the mapping process is the code generation discussed in Chapter 9 which creates the target simulations by inspecting this simulation tree representation.

The tree representations of each of the different simulation elements and how they are constructed from the DSL descriptions are discussed in Sections 8.2, 8.3 and 8.4. The algorithm used to combine and manipulate these trees are discussed in Section 8.5.

8.2 Equation Trees

The equation parser discussed in Chapter 7 can read equations written in ASCII in a mathematical form. When an equation is parsed the parser will construct a tree representation of it. This tree will contain all of the necessary information about the parameters, lattices, types and the mathematical expressions of the equations. The process of tree construction is incorporated into the ANTLR [147] parser which constructs the tree as it parses the equations.

Each equation tree contains three fixed nodes - Lattices, Parameters and Equations. Each of these contains multiple child nodes that store the necessary information about the model. Lattices and Parameters both contain a number of child parameter nodes which define a type and a name. Obviously the nodes in Lattices represent the lattices of the system while the nodes in Parameters are the adjustable parameters of the model.

The Equations node contains a set of trees which represents the equations of the model. Each tree consists of operator, parameter and stencil nodes which collectively form the mathematical expression of that equation. This tree structure can be used to represent the models read by the equation parser. Once again the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations are used as examples. The ASCII descriptions of these models were presented in Chapter 7 and their equation tree representations are shown in Figures 8.1, 8.2 and 8.3.

8.2.1 Cahn Hilliard Tree

The tree in Figure 8.1 contains all of the information necessary to define the Cahn-Hilliard model. The system is represented by a single lattice `u` which is of type `floating`. There are four adjustable parameters which can be used to control the model `M`, `B`, `U` and `K` which are all of type `floating`. Finally the actual Cahn-Hilliard equation is represented by the tree `d/dt u`.

The mathematical expression of the equation can be reconstructed by traversing this tree using in-order traversal. This traversal will produce the expression `M * Laplacian{-B * u + U*u*u*u - K * Laplacian{u}}`. The `Laplacian` node represents a stencil operator - in this case the Laplacian operator. These stencil nodes are discussed further in Section 8.3. This tree contains all of the necessary information to define the Cahn-Hilliard model within this simulation family.

8.2.2 Ginzburg-Landau Tree

The Ginzburg-Landau tree (Figure 8.2 can be constructed from its ASCII representation in the same way as the Cahn-Hilliard tree. It can be seen from this tree that the single lattice `u` is of type `complex` and that there are three parameters `P`, `Q` and `i` of type `complex` and one parameter `y` of type `floating`.

The equation also contains the operator node `ABS` which represents the mathematical expression |...| or absolute value. These functions must be supported by the target architecture complex number implementation in order for this simulation to be computed.

For a target code generator to produce an implementation of this model, it must support the type `complex`. A code generator can identify data types supported by its target language and use that

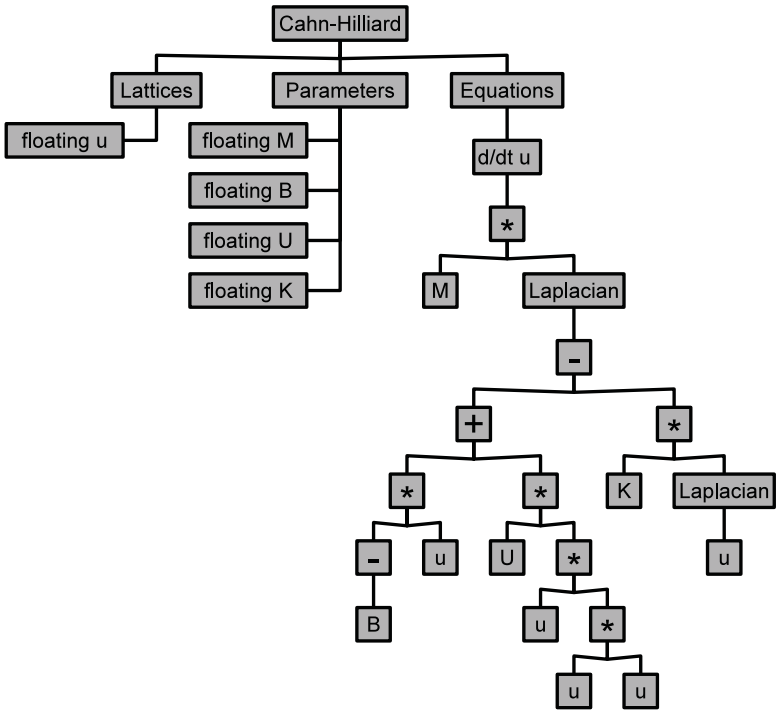


Figure 8.1: The STARGATES tree representing the Cahn-Hilliard equation created by the ANTLR equation parser.

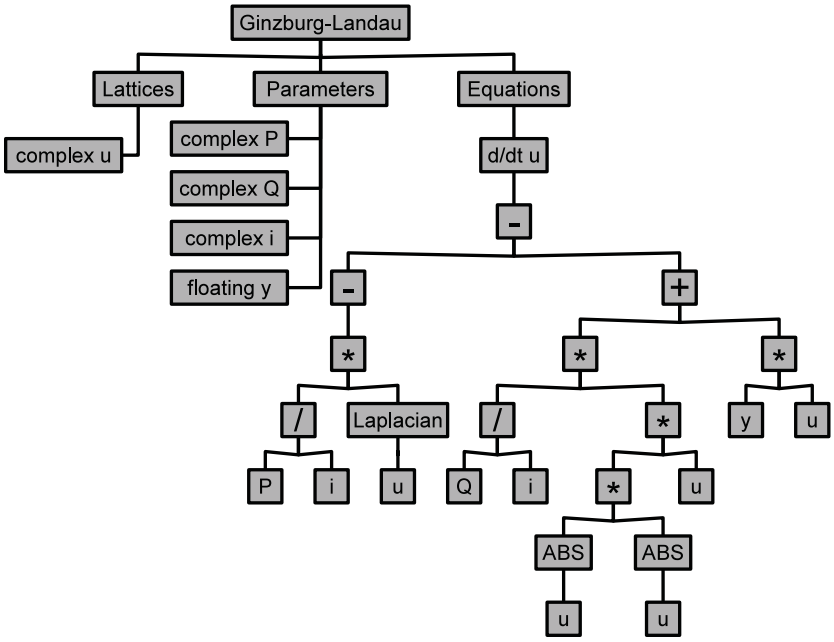


Figure 8.2: The STARGATES tree representing the Ginzburg-Landau equation created by the ANTLR equation parser.

native type. If a code generator does not recognise the data type it will assume it is a user-defined type that will be supplied to the final implementation. To execute a generated simulation of the Ginzburg-Landau equation, the user must supply a `complex` data type.

8.2.3 Lotka-Volterra Tree

Finally the Lotka-Volterra tree is shown in Figure 8.3. The main feature shown in this tree is the fact that the simulation state is stored by two lattices and governed by two equations. The two lattices representing the system are both of type `floating` and named `u0` and `u1`. There are six parameters of the model `A`, `B`, `C`, `D`, `D0` and `D1` which are all of type `floating`. The model is governed by the two equations which each compute the time derivative of one of the fields. These two equations are represented by the trees `d/dt u0` and `d/dt u1`.

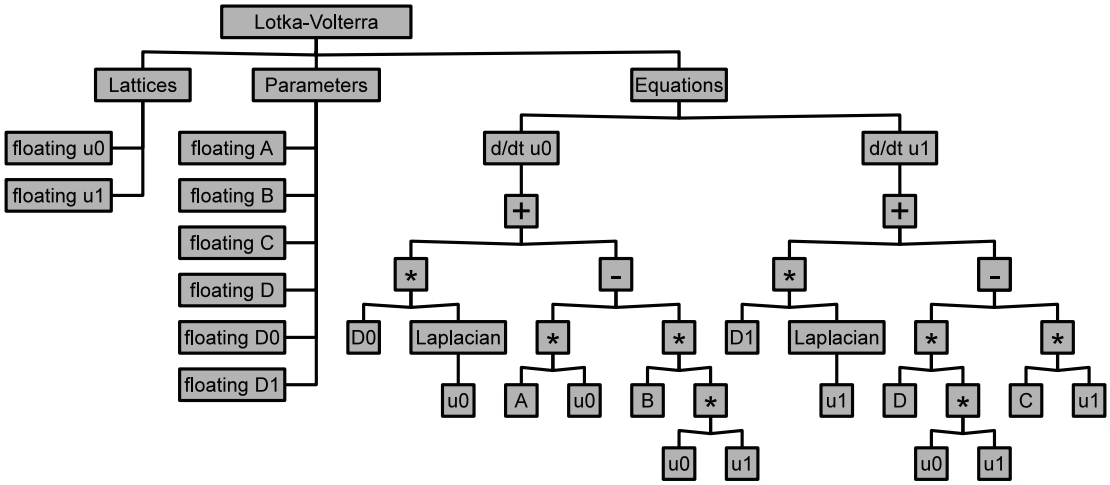


Figure 8.3: The STARGATES tree representing the Lotka-Volterra equation created by the ANTLR equation parser.

These trees contain all of the information required by STARGATES to define each of these three computational models. The ANTLR parser will construct the tree for a model as it parses the model's ASCII representation, provided that the description of the model conforms to the DSL discussed in Chapter 7. These trees only contain the information about a model. To produce a tree that contains all of the information required to generate a simulation, they must be combined with the other elements of the simulation. This combination process is discussed in Section 8.5.

8.3 Stencil Nodes

The equation trees discussed in Section 8.2 will almost always contain stencil nodes which represent the finite-difference operators used by the model. These Stencil nodes contain a finite-difference operator and a subtree of nodes. The stencil operators are implemented as a multi-dimensional array of expressions and the subtree represents the expression to which the operator is applied. This

operator will only affect the lattice nodes and subtrees containing lattice nodes in the stencil node's children.

The stencil array contains expressions which represent the coefficients of the operator. The expressions can be evaluated to specific values or left as null to represent an empty cell in the stencil. These coefficient expressions are parsed from the ASCII representation of the stencil discussed in Chapter 7. As an example, the arrays that represent the Laplacian stencil in two- and three-dimensions are shown in Figure 8.4. The cells containing the value 0 represent cells with a null expression.

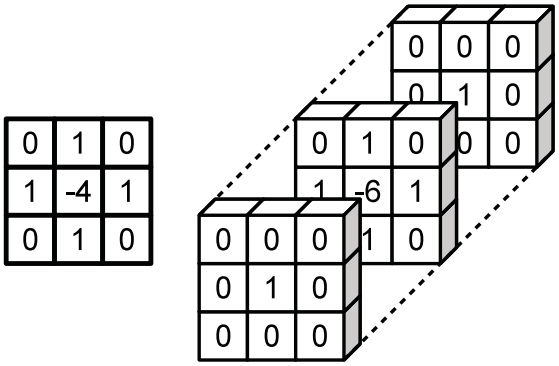


Figure 8.4: The arrays that represent the two- and three-dimensional Laplacian stencils. Values of 0 are used to represent empty cells which should not be included in the computation.

Once the equation trees have been constructed there are a number of ways the stencil nodes within the trees can be manipulated. These manipulations include both arithmetic and stencil-specific operations. The stencil library supports the arithmetic operations '+', '-', '*' and '/'. These stencil arithmetic operations simply involve applying the numerical operator to the corresponding values of the two stencil arrays. The centres of the two stencils are aligned and the values of any overlapping cells have the operation applied to them. The value of each cell in the resultant stencil is the result of this arithmetic operation. It is only valid to apply these stencil operations when both stencils contain identical subtrees.

The other way stencils can be manipulated is to apply one stencil to another. This operation allows equation trees to be rearranged to remove any nested stencils. Nested stencils are stencil nodes which belong to the subtree of another stencil node. Equation trees which contain nested stencils can be rearranged by applying the parent stencil to any stencil in its subtree.

This allows the function evaluation to be performed in a single computation rather than having to compute the equation in multiple stages. Another advantage of this operation is that it allows complex, higher-order stencils to be automatically constructed from simpler ones. One example of this situation is the Cahn-Hilliard equation given in Chapter 3 which is restated in equation 8.1.

$$\frac{\partial \mathbf{u}}{\partial t} = M\nabla^2\left(-B\mathbf{u} + U\mathbf{u}^3 - K\nabla^2\mathbf{u}\right) \tag{8.1}$$

This equation contains two Laplacian operators (∇^2) nested inside each other. This can also be seen in the tree created by parsing this equation (see Figure 8.1). One way to avoid these nested

stencils would be to define the equation with the stencils already expanded. This expanded form of the equation is shown in equation 8.2.

$$\frac{\partial \mathbf{u}}{\partial t} = M\left(-B\nabla^2\mathbf{u} + U\nabla^2\mathbf{u}^3 - K\nabla^4\mathbf{u}\right) \tag{8.2}$$

However this can make the equation definition more cumbersome and places the responsibility of expending operators on the user. While this method can still be used, STARGATES is capable of automatically combining these stencils to remove nesting. This can be performed by applying a stencil to any other stencil nodes in its subtree and inserting a stencil node at the head of any subtree which contains lattice nodes. Applying this algorithm will rearrange the Cahn-Hilliard equation tree shown in Figure 8.1 into the following tree (Figure 8.5). Note how the Laplacian node that was nested within the other Laplacian node has become a Laplacian^2 node

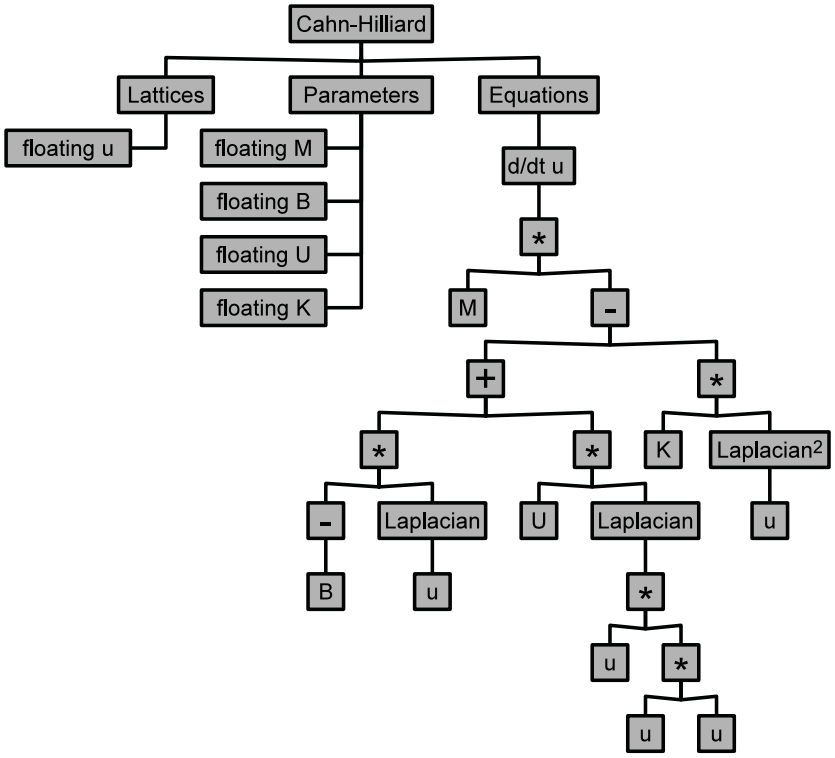


Figure 8.5: Tree representing the Cahn-Hilliard equation after it has been rearranged. Now no stencil node sub-tree contains another stencil node.

To convert the two Laplacian stencils into a Laplacian^2 stencil, the stencils are applied to each other. This can be performed numerically by applying one stencil to every cell of the other. This resultant stencil is the sum of the values of one stencil applied to each cell of the other and multiplied by the value of those cells. An example of this process in two- and three-dimensions for applying two Laplacian stencils to each other can be seen in Figure 8.6.

The stencil operators used by this system are not tied to any indexing scheme, rather they just

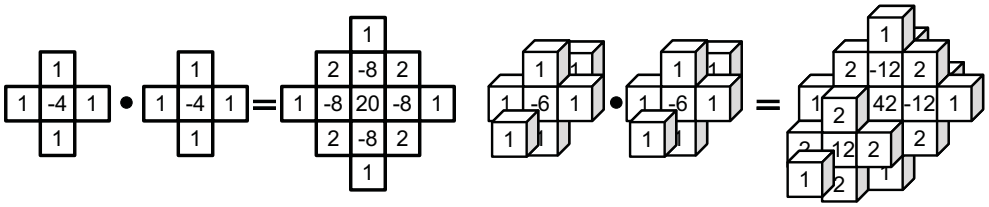


Figure 8.6: Two Laplacian stencils being applied to each other to create a Laplacian^2 stencil. Shown in two- and three-dimensions.

contain the information about the stencil size, dimensionality and actual values. Indexing schemes and access methods are defined purely in the target generator and thus the stencils remain independent of the target language.

The manipulation of these stencils and the stencil nodes within equation trees is part of the Generative Programming process of mapping problem definition to the target solutions. This assembly of simulation elements is possible because of the system's knowledge of the program family. It automates the process of expanding stencils and equations that would otherwise have to be performed manually.

8.4 Integration Trees

Chapter 7 describes how any explicit Runge-Kutta integration method can be defined using a simple table of coefficients known as a Butcher tableau [62]. A developer can easily define an integration method by writing the coefficients in an ASCII form. Tableaux can be of any size which determines the number of stages of the integration method. Because STARGATES is currently limited to using explicit Runge-Kutta methods, the tableau must be lower triangular to ensure the method is explicit (see Chapter 3). This is a very simple way of defining an integration method as there is no need to write out any mathematical expressions.

When an ASCII tableau is parsed, STARGATES will construct a tree representing the integration method. These integration trees are very similar in nature to the equation trees discussed in Section 8.2. The integration methods are stored as trees so that they can be manipulated and combined together with an equation tree to form a single tree that represents the entire simulation. This method of merging integration and equation trees is discussed in Section 8.5.

To construct an integration tree, STARGATES must determine the parameters of the method, the lattices required to store the stages of the method and the actual mathematical expressions to compute the integration. This is a similar set of information to the information stored in an equation tree. To automatically construct an integration tree from a Butcher tableau, all this information must be determined by inspecting the size and coefficients of the tableau.

The explicit Runge-Kutta integration methods used in this research use a fixed time step to integrate the system. All the methods use a single parameter to control the time step which can be automatically inserted into an integration tree as it is created. This time step parameter is named h for all the integration trees. To implement the adaptive stepsize methods discussed in Appendix A,

another parameter e could be inserted to control the maximum allowable error.

Secondly the lattices required by the integration method must be determined. Every integration method requires an input and output lattice (Y_0 and Y_s) and may require multiple extra lattices to store the intermediate stages. The number of required lattices can be found easily by counting the number of rows in the tableau.

Finally the mathematical expressions must be constructed that compute the values of the intermediate stages and the final output value. The expressions can be constructed from $Y_i = Y_0 + h \times (\sum_{j=0}^{i-1} a_{i,j} F_j)$ and $Y_s = Y_0 + h \times (\sum_{i=0}^{s-1} b_i F_i)$ which can be converted to a tree representation. The algorithm that constructs these integration trees is shown in Algorithm 1.

Algorithm 1 Pseudo code for constructing a STARGATES integration tree from a Butcher tableau.

create Head node
add Lattices, Parameters and Steps nodes to Head node
for all i in rows **do**
  add Y_i node to Lattices node
end for
add h to Parameters node
for i in $0..s-1$ **do**
  add $step_i$ to Steps node
  create $integration\_evaluations$
  for j in $0..i-1$ **do**
    if $(a[i][j]! = 0)$ **then**
      add $+a[i][j] \times F_j$ to $integration\_evaluations$
    end if
  end for
  create $integration\_stage$
  $integration\_stage \leftarrow Y_i = Y_0 + h \times (integration\_evaluations)$
  add $integration\_stage$ to $step_i$
end for
add $step_s$ to Steps node
create $integration\_evaluations$
for i in $0..s-1$ **do**
  if $(b[i]! = 0)$ **then**
    add $+b[i] \times F_i$ to $integration\_evaluations$
  end if
end for
create $integration\_stage$
$integration\_stage \leftarrow Y_s = Y_0 + h \times (integration\_evaluations)$
add $integration\_stage$ to $step_s$

At this point the integration method does not contain any information on data types. This data type information will be substituted into the integration trees when they are merged with equation trees. It should also be noted that at this point the tableau coefficients need not be evaluated to a specific value. The parsed expression for each coefficient can be inserted into the tree for each term of the integration method. Two examples of constructing integration trees from their tableau representations are given in the following sections.

8.4.1 Euler

The Euler method is rarely used because it is generally unstable for most simulations. However, it is still useful as an illustrative example to show how an integration method can be generated from its Butcher tableau representation. The Euler method is a single-stage integration method which can be represented in ASCII as a Butcher tableau as:

```
0 | 0
b | 1
```

Applying Algorithm 1 to this tableau will produce an integration tree representing the Euler integration method. The lattices required for the integration method are simply the input and output lattices Y_0 and Y_1. The tableau only has these two rows and thus no other intermediate lattices are required. These two lattices will be created and added to the subtree of the Lattices node. The time step parameter h is added to the Parameters node of the integration tree. Finally the mathematical expression to compute the integration method is formed.

The first row of the tableau represents the input lattice Y_0 which is the input to the integration method. The second row b is computed using $Y_s = Y_0 + h \times (\sum_{i=0}^{s-1} b_i F_i)$. There is only one column in this tableau and substituting the coefficients into this gives $Y_1 = Y_0 + h \times (1 \times F_0)$.

When this expression is constructed as a tree it can be inserted into the integration tree as the single stage of the method. The tree representation of this simple integration method is constructed by STARGATES and results in the tree shown in Figure 8.7.

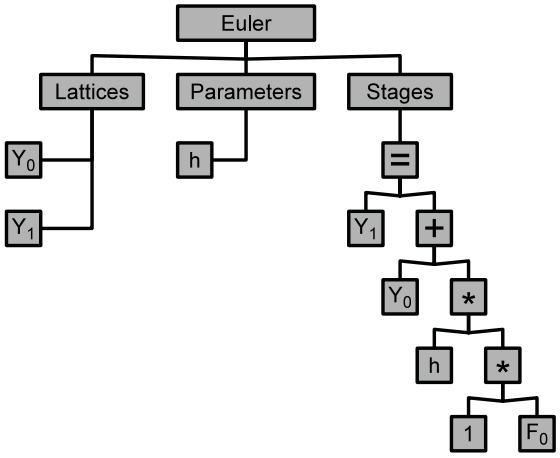


Figure 8.7: The tree constructed by the Integration Library that represents the Euler integration method.

8.4.2 RK2

The same process can be used to construct trees for higher-order integration methods from their Butcher tableau representations. The Runge-Kutta 2nd order method is a two-stage integration

method which has been discussed in Chapter 3. This integration method can be defined as an ASCII tableau as follows:

```
0   | 0   | 0
1/2 | 1/2 | 0
b   | 0   | 1
```

This fixed time step integration method has a single parameter h to control the time step. The three rows in the tableau represent the input/output lattices Y_0 and Y_2 as well as an intermediate stage Y_1. Finally the two computational stages are constructed using the coefficients $\frac{1}{2}$ and 1 and the function evaluations of the input and intermediate stages F_0 and F_1. From the ASCII representation of this integration method, STARGATES can automatically construct the integration tree in Figure 8.8 by applying Algorithm 1.

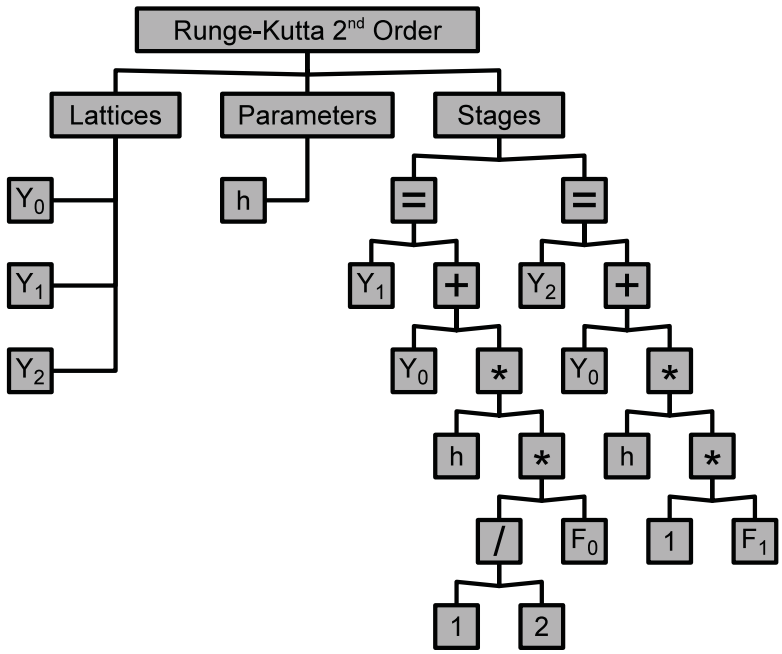


Figure 8.8: The integration tree that represents the Runge-Kutta 2^{nd} order integration method.

From these two examples it should be possible to see how higher-order integration methods can be constructed from tableaux. These trees can quickly become very large and cumbersome to describe (See Appendix A for examples), constructing integration trees automatically from Butcher tableaux makes these higher-order methods more accessible.

8.5 Merging Equation and Integration Trees

Equation and integration trees must be merged together to form a single tree representing the simulation. It is this tree that represents all the computation required to numerically integrate the equation

over time using the specified integration method. Some complexities can arise in this process, especially when dealing with complex integration methods and models that have multiple lattices and governing equations.

The form of a simulation tree is again very similar to the form of the equation and integration method trees. The simulation has a set of lattices, a set of parameters and a set of mathematical expressions. These are represented by three nodes Lattices, Parameters and Steps. The subtrees of these nodes must be determined by examining both the equation and integration trees and merging them appropriately.

The parameters of a simulation are found easily by combining the parameters of the equation and the integration trees. All the integration methods currently supported use a fixed time step controlled by a single parameter h. However, this tree structure could work with other integration methods that may have other controlling parameters. One example of this would be an adaptive stepsize method that uses an error term to control the time step of the integration method. The equations may also have any number of parameters to control the behaviour of the system. If the equation tree contains a parameter named h, the time step parameter for the integration method will be renamed to ensure there are no naming conflicts in the final simulation.

Determining the required lattices for a simulation is somewhat more complex than simply combing the equation and integration lattices. The lattices in the integration method represent intermediate system states that must be stored to compute the integration method. The equation lattices represent the lattices necessary to store a single system state. The total number of lattices required by a simulation is the number of integration method system states times the number of lattices in the equation. The simulation lattices are named by substituting the name of each equation lattice to replace the Y in each system state Y_i belonging the integration method. For instance, an integration method with system states Y_0 and Y_1 combined with an equation with a single lattice u with form a simulation tree with the lattices u_0 and u_1.

The mathematical expressions which represent the integration of the equation over time are constructed by substituting the equation trees into the integration stage F_i nodes. The target of each F_i node is Y_i which identifies the system stage that should be used in the computation. To evaluate the equation for that system stage all the lattice nodes in the equation tree will be replaced with that lattice for that stage. One copy of each stage of the integration method will be used to integrate each equation governing the system. The process for merging these trees is presented in pseudo-code in Algorithm 2.

The algorithm in Algorithm 2 can be used to combine the equations and integration method trees to form a single simulation tree. The tree contains all of the information about the equations, stencils and integration method used by the simulation. The following sections give some examples of how integration and equation trees can be combined using this algorithm. The combinations used as examples are - Cahn-Hilliard & Euler, Cahn-Hilliard & RK2 and Lotka-Volterra & RK2.

8.5.1 Cahn-Hilliard Euler

Figure 8.9 shows how the Cahn-Hilliard equation tree from Figure 8.1 can be combined with the Euler integration tree from Figure 8.7. The integration method lattices Y_0 and Y_1 have been combined

Algorithm 2 Pseudo code for inserting Equation trees into a STARGATES integration tree.

```
create Head node
add Lattices, Parameters and Steps nodes to Head node
for all Y_i in Integration Lattices do
  for all l in Equation Lattices do
    add l_i to Lattices
  end for
end for
for all p in Equation Parameters do
  add p to Parameters
end for
for all p in Integration Parameters do
  if p exists in Parameters then
    rename p
  end if
  add p to Parameters
end for
n = 0
for all integration_stage in Integration Stages do
  add step_n to Steps
  for all equation in Equation Equations do
    copy integration_stage to calculation
    for all node in calculation do
      if node == F_i then
        copy equation into node
        for all lattice in Equation Lattices do
          replace lattice in node with lattice_i
        end for
      end if
    end for
    add calculation to step_n
  end for
end for
```

with the equation lattice `float` u into two lattices `float` u_0 and `float` u_1. The node F_0 from the integration method has also been replaced with the Cahn-Hilliard tree and all the u nodes from the equation tree have been replaced with u_0.

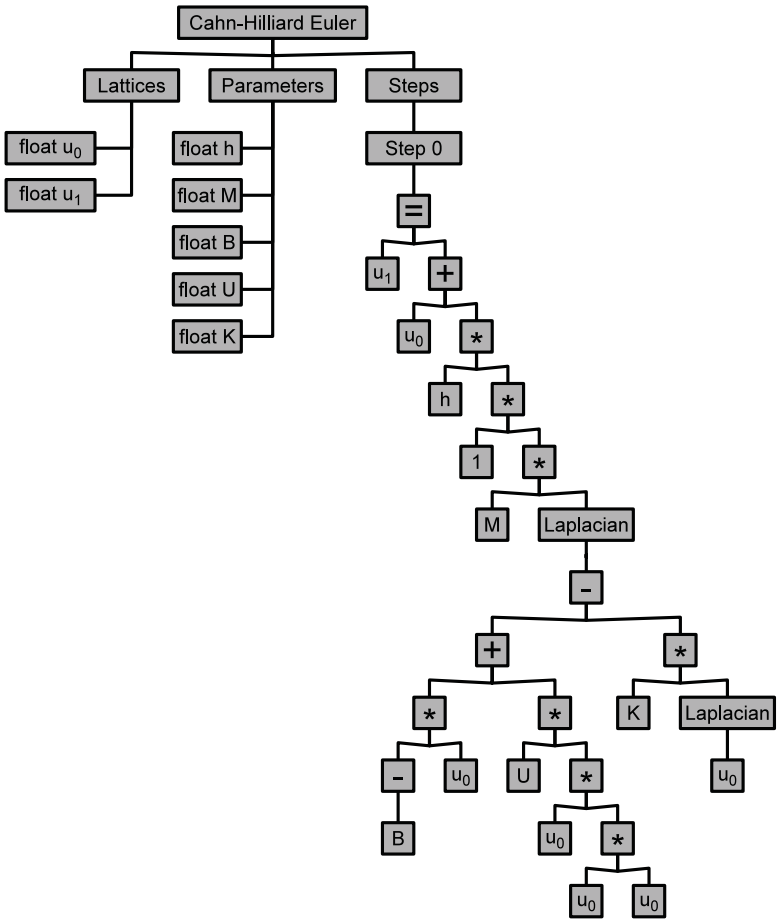


Figure 8.9: The simulation tree representing a simulation of the Cahn-Hilliard equation using the Euler integration method.

This is a simple example because the Euler integration method only has a single integration step and the Cahn-Hilliard equation has a single lattice and a single governing equation. A more complicated example is combining the Cahn-Hilliard equation with the RK2 method.

8.5.2 Cahn-Hilliard RK2

The RK2 integration is one of the most simple Runge-Kutta methods that uses intermediate lattices in the final computation. The integration tree of this method can be seen in Figure 8.8 and tree representation of the combination of the Cahn-Hilliard equation and this integration method can be seen in Figure 8.10.

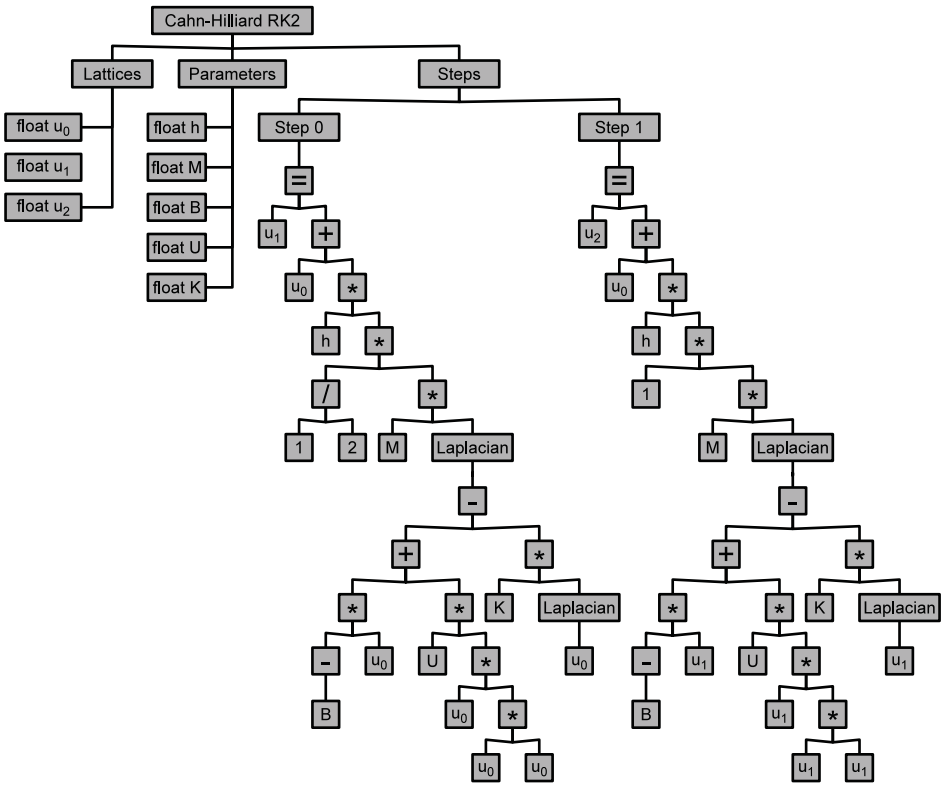


Figure 8.10: The simulation tree representing a simulation of the Cahn-Hilliard equation using the RK2 integration method.

This method now has three lattices, `float` u_0, `float` u_1 and `float` u_2 which represent the initial system state, the computed intermediate stage and the final system state after the time step. The tree also has two calculation steps the first of which computes $u_1 = u_0 + h \times (\frac{1}{2} \times F_0)$ and the second of which computes $u_2 = u_0 + h \times (1 \times F_1)$. Note that in the tree representing the first step u_0 has been substituted into the Cahn-Hilliard equation tree and the second step has substituted u_1. The information about which lattice to substitute is gathered from the nodes in the integration method. The first step has the node F_0 then this lattice u_0 is substituted into the equation tree and replaces all the u nodes. Likewise the second step has a node F_1 then u becomes u_1 and this is the lattice which is substituted into the equation for the second step.

8.5.3 Lotka-Volterra RK2

The Lotka-Volterra model introduces an interesting complication, the tree representing this model has two lattices and two governing equations. Combining this model with an integration method tree requires some additional lattices and equation substitutions to be made. For reference the Lotka-Volterra equation tree can be found in Figure 8.3.

To compute an update for a model like the Lotka-Volterra equation with two lattices and two coupled equations, two computation steps are required to calculate a new integration stage. When a model such as this is combined with a multi-step integration method like the RK2 method, there will be two calculations necessary per integration step and two lattices will be necessary to store each intermediate state. The combination of the Lotka-Volterra model with the RK2 integration method can be seen in Figure 8.11.

Each step of the integration method now has to compute two separate calculations, one for the $u0$ lattice and one for the $u1$ lattice. The two lattices of the equation have been combined with the necessary stages of the integration method which results in six separate lattices. These lattices represent the system before the time step ($u0_0$ and $u1_0$), the intermediate system ($u0_1$ and $u1_1$) and final system after the time step ($u0_2$ and $u1_2$). This simulation tree is significantly more complicated that the single equation Cahn-Hilliard examples. These trees become increasingly large for complex models with more governing equations and high-order integration methods which require more stages.

8.6 Conclusions

This chapter has discussed how simple ASCII representations of field equations, discrete finite-differencing stencils and explicit Runge-Kutta methods can be converted into internal data structures and manipulated. The stencils are stored as multi-dimensional arrays while the equations and integration methods are converted into tree structures. These trees can be manipulated and combined to form a single tree representing the simulation.

This process is the first stage of mapping from the problem space definition to the solution space implementation. This part of the Generative Programming process relies on knowledge of the program family to combine the different elements together. The second part of mapping to a solution implementation is the code generation stage that traverses this simulation tree and produces target-specific simulation code. This simulation tree investigation and code generation phase is discussed in the following chapter - Chapter 9.

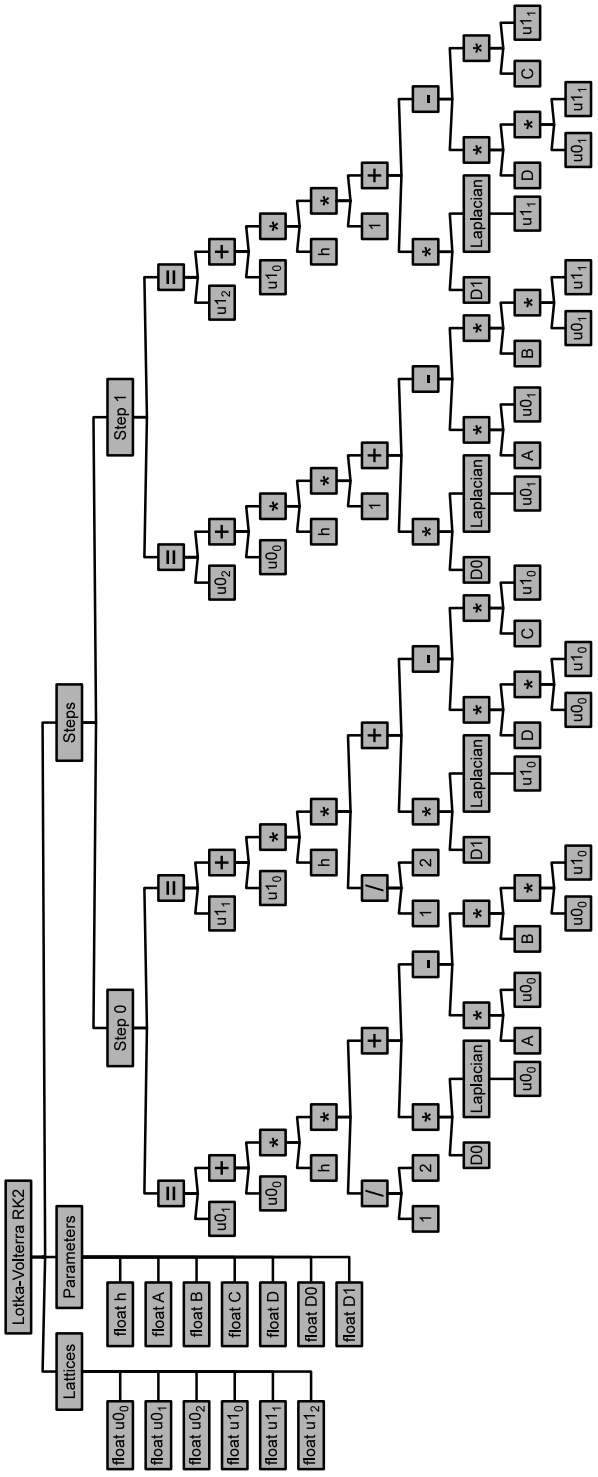


Figure 8.11: The simulation tree representing a simulation of the Lotka-Volterra equation using the RK2 integration method.

"I don't think necessity is the mother of invention - invention, in my opinion, arises directly from idleness, possibly also from laziness. To save oneself trouble."

Agatha Christie

9 Automatic Code Generation

9.1 Introduction

Chapter 7 described a DSL for defining the elements of finite-differencing simulations of partial differential field equations using explicit Runge-Kutta integration methods. Chapter 8 discussed how these elements can be stored as trees and manipulated and merged to form a single tree representing the simulation. This chapter now considers how parallel simulation code can be generated from these abstract trees through the use of target-specific code generators. All of the implementation, language and architecture specific information is stored within these target code generators. This allows the abstract simulation tree to be completely separate from any implementation-specific information. This represents the second part of the Generative Programming process where the definition of the simulation is mapped into solution space.

Each generator effectively defines its own template of a simulation implementation. The generator will populate this template using a number of patterns. Generators make a number of assumptions about the manner in which the simulations can be constructed. This is a valid approach to generating parallel simulations due to the fact that STARGATES is limited to producing solutions for a specific program family. The details of the simulations such as the actual computation, memory allocation, communication etc will be filled into this template using the patterns and information from the simulation tree.

The generators must traverse the simulation tree and extract from them the information required to generate the target code to compute the simulation. There will be many similarities between the different generators and the manner in which they traverse the tree but the generators should remain distinct from one another. These generators must make all decisions on the data-types, precision and parallel algorithm used to compute the simulation.

Different generators can be used to generate target code that uses different update algorithms or implementations, or a single generator can have the option to generate simulations that use different options and optimisations. Another attractive feature that can be built into these code generators is the use of heuristics to automatically include optimisations within the produced simulation code. These heuristics must ensure that these optimisations are both valid and applicable.

The advantage of this approach is that the front-end parsing and simulation tree construction for the simulations remains the same regardless of the output generator used. When a new architecture or language is released, a new generator can be written that will allow an entire simulation base to be migrated to make use of that new architecture or language. This makes it much easier to adopt a new language or architecture without the need to rewrite the entire simulation base. This is a far easier and more extensible programming model than maintaining separate code versions for each simulation and architecture.

9.2 Equation Generation

One very similar element between all of the code generators is the generation of the code to perform an equation evaluation. The syntax for fetching values from memory will vary between different target generators, however the actual calculation string will be very similar if not identical for many of them. The computation string can be easily generated by performing an in-order traversal of the tree and outputting each node as appropriate. The Cahn-Hilliard equation is used as an example; the simulation tree for a Cahn-Hilliard simulation using the Euler method is shown in the previous chapter in Figure 8.9. When this tree is traversed using an inorder traversal, the output string produced will be:

```
u_1 = u_0 + M*((-B*Laplacian(u_0) + U*Laplacian(u_0*u_0*u_0))-K*Laplacian 2(u_0))
```

Each of the parameters M, B, U and K can be directly referenced by its name but the lattice and stencil nodes are somewhat more complicated. Different languages will have different indexing and addressing schemes; even within the same language there are sometimes multiple ways to create and index arrays. Because each cell in the lattice must be updated, each of the nodes in the equation tree corresponding to a lattice will be replaced with the lattice name and the index into that lattice using an appropriate indexing scheme. The index values and scheme will depend on the specifics of that code generator and the algorithm it is implementing.

The code produced to apply a stencil to a lattice depends on the dimensionality of the simulation being generated. The output generator will examine the stencil matrix to determine which neighbouring values must be used in the calculation and what weights should be applied to them. Using the index notation of u[x,y], the stencil `Laplacian(u)` (see Chapter 8 for the Laplacian stencil matrix) will be expanded in two dimensions to:

```
u[x,y-1]+u[x-1,y]-4*u[x,y]+u[x+1,y]+u[x,y+1]
```

or in three dimensions to:

```
u[x,y,z-1]+u[x,y-1,z]+u[x-1,y,z]-6*u[x,y,z]+u[x+1,y,z]+u[x,y+1,z]+u[x,y,z+1]
```

Using these techniques, the code to compute the update of a single cell in the lattice can be constructed by substituting these stencil expansions into the stencils in the output string. All the decisions on precision, syntax and semantics of the equation calculation are made by the code generator during this step. The tree is traversed by recursively processing each node in the tree. Each node in

the tree has a member $value$ which is the contents of the node and two children $left$ and $right$ which may or may not be used. The recursive algorithm for generating code by traversing an equation tree is given in Algorithm 3.

Algorithm 3 Pseudo code for generating equation code from a tree.

```
process(Node n)
if n is a parameter then
  generate value
else if n is a constant then
  generate value
else if n is a operator then
  if left ≠ null then
    generate "(" process(left) ")"
  end if
  generate value
  generate "(" process(right) ")"
else if n is a lattice then
  determine [index] based on index scheme and stencil
  generate value[index]
else if n is a stencil then
  array ← value
  for all cell in array do
    if cell ≠ 0 then
      generate cell "*" "(" process(right) ")"
    end if
    if not first value then
      generate "+"
    end if
  end for
end if
```

The application of this algorithm to the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equation trees can be used to generate the code to compute the equations. The exact form of the algorithm given in Algorithm 3 will produce equation code using infix notation and standard operators. This is suitable for C-based languages such as those discussed in Chapter 4. However, alternative algorithms can easily be developed to produce code for any language or notation scheme.

9.2.1 Example Generated Equation Code

The following code listings 9.1, 9.2 and 9.3 show the C-syntax code generated by applying the algorithm described in Algorithm 3 to the equation trees of the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra models. These equation trees are given in Chapter 8 and have been constructed from the ASCII representations of the models given in Chapter 7.

First the generated code for the Cahn-Hilliard model is shown in Listing 9.1. The example shows the calculation of the Cahn-Hilliard model in two-dimensions and the naming convention `u0yx` is used to show the value fetched from the lattice `u0` at position `(x,y)`. The code can be compared to the hand-written code for the same model given in Chapter 5. This automatically generated code

has more parentheses which makes it slightly less reader-friendly, but this could be improved with relatively minor changes to the generator.

Listing 9.1: Generated C code that calculates the Cahn-Hilliard equation.

```
(((M*((((-B)*( (u0ym1x) +
   (u0yxm1) + (-4*u0yx) + (u0yxp1) +
                (u0yp1x)))
+(U*(                        ((u0ym1x*u0ym1x)*u0ym1x) +
((u0yxm1*u0yxm1)*u0yxm1) + (-4*(u0yx*u0yx)*u0yx)  + ((u0yxp1*u0yxp1)*u0yxp1) +
                           ((u0yp1x*u0yp1x)*u0yp1x))))
-(K*(                        (u0ym2x)  +
          (2*u0ym1xm1) + (-8*u0ym1x) + (2*u0ym1xp1) +
(u0yxm2) + (-8*u0yxm1) +  (20*u0yx)  + (-8*u0yxp1)  + (u0yxp2) +
          (2*u0yp1xm1) + (-8*u0yp1x) + (2*u0yp1xp1) +
                           (u0yp2x))))))
```

Listing 9.2 shows the code produced by STARGATES that computes the Ginzburg-Landau model. This particular code listing computes the Ginzburg-Landau model as a single equation of type complex. It assumes that a type `complex` will be provided with support for the operations `+`, `-`, `*`, `/` and `abs()`.This code fragment can also be compared to the hand-crafted version in Chapter 5.

Listing 9.2: Generated C code that calculates the Ginzburg-Landau equation.

```
(((((-(P/i))*(  (u0ym1x) +
    (u0yxm1) + (-4*u0yx) + (u0yxp1) +
               (u0yp1x)))
-((((Q/i)*(abs(u0yx)))*(abs(u0yx)))*u0yx))
+ (y*u0yx)))
```

Finally the generated code to compute the Lotka-Volterra model is given in Listing 9.3. This code example has two lattices `u00` and `u01` that are used in the calculation of both of the governing equations.

Listing 9.3: Generated C code that calculates the Lotka-Volterra equation.

```
(((((A*u00yx)-
((B*u00yx)*u10yx))+
(D0*(              (u00ym1x) +
     (u00yxm1) + (-4*u00yx) + (u00yxp1) +
                 (u00yp1x)))))

((((((C*u00yx)*u10yx)-
(D*u10yx))+
(D1*(              (u10ym1x) +
     (u10yxm1) + (-4*u10yx) + (u10yxp1) +
                 (u10yp1x)))))
```

This type of equation tree traversal can be used to produce code to compute any equation that can be described using an equation tree. The code for computing the three different equations has been produced automatically from simple ASCII descriptions of the models. These generated code listings of the three equations act as proof of concept that code for different computational models can be automatically generated from simple descriptions.

9.3 Generating Target Specific Code

Generating the rest of the simulation is far more architecture specific than generating code for the equation calculation. Not only will the exact syntax and indexing schemes be different, but the very nature of the update algorithms will change. Four different code generators are used as examples that encompass all of the methods of parallelism discussed in Chapter 5. These generators produce simulations using C, TBB, MPI and CUDA. Other simulations can be constructed using hybrids of these languages/libraries but these are not discussed here.

The generators produce code that uses C syntax but it should be noted that this is simply because of preference and simplicity of comparison - it is not a limitation of the generator. These examples are used to show how code of very different paradigms can easily be generated from the same input simulation tree.

The generators have a template simulation code that they populate with patterns and details extracted from the simulation tree. These templates are not always fixed and different configurations can be used depending on the configuration of the simulation and according to any heuristic-based optimisations built into the generators. Generator developers have a lot of freedom to modify and construct generators however they want. Knowledge of the nature of the target simulations and parallel programming is vital to constructing an efficient and correct parallel code generator.

9.4 Generating C code

The C generator produces single-threaded C implementations to compute the simulations of models. The generator must consider the memory allocation, initialisation of parameters and lattices required by the simulation. It uses the method of traversing the simulation tree to produce code that calculates the equation as discussed in Section 9.2.

A similar process of traversing the tree is necessary to determine the required lattices and parameters of the model and their data types. The initialisation of the model lattices and parameters depends also on the configuration of the simulation. This configuration provides information such as the dimensionality of the simulation, the values of the parameters, initialisation of lattices etcetera.

At this point in the construction process the exact data types of the simulation parameters and lattices will be defined. Parameters defined in the simulation tree as type `floating` will be implemented as type `double` by default but can be configured to use `float` or some other precision floating point value if desired.

9.4.1 C Template

The algorithm shown in Algorithm 4 is one of the possible templates for generating C code to compute a finite differencing simulation. The simulation code produced by this generator will initialise the parameters and lattices in the main function and integrate the system over time using a single update function. This update function will compute the new value of the system after the time step using the integration method defined by the simulation tree.

Algorithm 4 C code generator template for a finite differencing simulation.

```
generate includes

generate update function
for all stage in Steps do
  generate iteration code
  generate neighbour access code
  for all equation in stage do
    traverse equation to generate equation code
  end for
end for

MAIN
generate parameter allocation
generate parameter initialisation
generate lattice allocation
generate lattice initialisation code
generate time step iteration code
generate function call
generate end iteration code
```

The update function will compute a new system state by iterating over the lattice in each dimension and computing the new value for each cell using the code generated by traversing the equation trees. These calculated values will be written to another lattice which may be the state of the system after the time step or an intermediate stage. The function may have to iterate over all the lattice cells multiple times depending on the number of stages of the integration method.

To generate the elements given in this template, the generator will make use of a number of patterns. These patterns are populated with details taken from the simulation tree and form the actual code implementation of the template. The C code generator patterns are discussed in the following section.

9.4.2 C Patterns

There are a number of patterns that the generator will use to produce these generated code segments. The patterns are written in a regular form here for ease of comprehension. In these patterns $\leftarrow$ represents an output operator; everything after this operator is generated code that will be returned by the pattern. Items in **bold** font represent variables or functions internal to the generator. Three important patterns are described here - they produce code for allocate memory, index into a lattice and iterate over a lattice.

To allocate the memory required for the lattices used by the simulation, the generator must determine the size, data type and name of each lattice. Names and data types can easily be determined by investigating the children of the Lattice node in the simulation tree. The size can be determined by querying the configuration file given to the generator. The dimensionality and size of the desired simulation is given in the configuration file in the form **size** = **X**$\times$**Y**$\times$**Z**$\times\ldots$. The number of integers determines the dimensionality of the simulation and the actual values give the size of the system

in each dimension. This information can be substituted into the pattern in Algorithm 5 to allocate memory for a lattice.

Algorithm 5 C Generator - lattice allocation pattern.

```
← type *name = (type*)malloc(size*sizeof(type));
```

To construct an index for this type of array, multiple indexes for each dimension must be combined into a single index. This requires the use of an array **dim** which contains the names of the dimensions. The **dim** array is constructed from the configuration input **size**. An internal variable **D** is used which stores the length of this array **dim**. An array **index** which contains the names of indexes in each dimension can be used to construct a single index using the pattern in Algorithm 6. Listing 9.4 shows an example of the indexes this pattern creates.

Algorithm 6 C Generator - index calculation pattern.

```
for i = D-1..0 do
  if i <D-1 then
    ← +
  end if
  ← index[i]
  for d = i-1..0 do
    ← *dim[d]
  end for
end for
```

The code to iterate over a lattice is somewhat more complex as the iteration can be done in a number of ways. The example implementations given in Chapter 5 use multiple `for` loops to iterate over a lattice, thus this method of iteration is used. The number of `for` loops required will be determined by the dimensions of the system that are stored in an array **dim** and its length **D**. The pattern shown in Algorithm 7 is used to construct a set of `for` loops to iterate over the lattice in each dimension - an example of the `for` loops constructed using this pattern can be seen in Listing 9.4.

Algorithm 7 C Generator - lattice iteration pattern.

```
for d = D-1..0 do
  ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
end for
```

These patterns should be familiar to any programmer with a knowledge of the C programming language. This process of constructing simulations from templates and patterns is an automation of the process a programmer would perform when writing an application. This particular template and set of patterns is only one of the possible ways to implement a simulation using the C-programming language. These generators can easily be configured to use different indexing schemes, iteration loops etcetera.

9.4.3 Generated C code

Code produced by the C code generator can be seen in Listing 9.4. This code fragment shows the C code to allocate memory for lattices and iteratively call an Euler update method to integrate the system over time. The update method iterates over the lattices using two loops and computes the update for each lattice cell.

Listing 9.4: Generated C code for a two-dimensional finite-differencing simulation using Euler integration. `euler` updates the system lattice with one single stage.

```
void euler(double *u0, double *u1, double h) {
  for(int iy = 0; iy < Y; iy++) {
    for(int ix = 0; ix < X; ix++) {
      double u0yx = u0[iy*X + ix];
      //compute equation for cell ix,iy
      //u1 = u0 + f(u0) * h
    }
  }
}

int main(int argc, char **argv) {
  double *u0 = (double*) malloc(Y*X*sizeof(double));
  double *u1 = (double*) malloc(Y*X*sizeof(double));
  for(int t = 0; t < 1024; t++) {
    euler(u0,u1,h);
    swap(&u0, &u1);
  }
}
```

9.5 Generating TBB code

The TBB generator produces code to compute the simulation in parallel using the Threading Building Blocks [111] libraries. This generator is designed to construct a TBB implementation based on the hand-written simulation described in Chapter 5. TBB is C-syntax based and so many parts of the generator will be identical to the C generator. The main difference with this implementation is the code used to iterate over the lattices.

9.5.1 TBB Template

The TBB template shown in Algorithm 8 is very similar to the C generator template from Algorithm 4. The main function will initialise the parameters and the lattices of the simulation. It also iterates over time and calls the update function which computes a single time step.

The update function of the TBB implementation will not use `for` loops like the C generator, instead it will use the TBB construct `parallel_for`. The workings of this TBB construct is discussed in Chapter 5. The patterns to construct these `parallel_for` calls are discussed in the following section.

Algorithm 8 The code template for generating a TBB finite-differencing simulation. This generator creates one function for each integration step and uses the `parallel_for` function to iterate over the cells in parallel.

```
generate includes

generate update function
for all stage in Steps do
  generate TBB parallel iteration code
  generate neighbour access code
  for all equation in stage do
    traverse equation to generate equation code
  end for
end for

MAIN
generate parameter allocation
generate parameter initialisation
generate lattice allocation
generate lattice initialisation
generate time step iteration code
generate function call
generate end iteration code
```

9.5.2 TBB Patterns

The TBB generator only requires three new patterns. To construct the rest of the simulation it will use the same patterns described in Section 9.4. The new patterns create TBB code to iterate over a lattice. The three patterns create TBB structures to partition the lattice, TBB loops to iterate over the lattice in parallel and code to calculate an index within these loops.

The lattice is partitioned using the TBB construct `blocked_range` for a one-dimensional lattice, `blocked_range2d` for a two-dimensional lattice and `blocked_range3d` for a three- or higher-dimensional lattice. This requires the generator to have dimension specific behaviour and three different patterns to construct this code.

This is one example of how specific optimisations can be included in the output generators. The pattern to construct these three different structures based on the number of dimensions **D**, splitting the lattice into block of size **grain** is given in Algorithm 9.

Algorithm 9 TBB Generator - lattice partition pattern.

```
if D==1 then
  ← blocked_range<int>(0, dim[0], grain[0])
else if D==2 then
  ← blocked_range2d<int>(0, dim[1], grain[1], 0, dim[0], grain[0])
else
  ← blocked_range3d<int>(0, dim[2], grain[2], 0, dim[1], grain[1], 0, dim[0], grain[0])
end if
```

The code used to iterate over these blocks can also be constructed using a pattern with dimen-

sion specific behaviour. This pattern will construct one loop for each dimension of the simulation. The `for` loops will iterate over the block for the first three dimensions and over the entire lattice length for any higher dimensions. This code assumes some `blocked_range` structure name `range` is provided. Algorithm 10 shows the pattern to construct the iteration and index calculation code.

Algorithm 10 TBB Generator - iteration and index calculation pattern.

```
for d = D-1..0 do
  if d==0 then
    ← int dim[0]b = range.cols().begin();
    ← int dim[0]e = range.cols().end();
    ← for(int idim[0] = dim[0]b; idim[0] <dim[0]e; idim[0]++) {
  else if d==1 then
    ← int dim[1]b = range.rows().begin();
    ← int dim[1]e = range.rows().end();
    ← for(int idim[1] = dim[1]b; idim[1] <dim[1]e; idim[1]++) {
  else if d==2 then
    ← int dim[2]b = range.pages().begin();
    ← int dim[2]e = range.pages().end();
    ← for(int idim[2] = dim[2]b; idim[2] <dim[2]e; idim[2]++) {
  else
    ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
  end if
end for
```

Finally the pattern for the actual `parallel_for` call is needed. The pattern is shown in Algorithm 11 and makes use of the previous two patterns to construct the parallel iteration over the lattice. The code produced by this particular `parallel_for` pattern will use lambda functions.

Algorithm 11 TBB Generator - lattice iteration pattern.

```
if D==1 then
  ← parallel_for(blocked_range, [=](blocked_range<int> &range) {
    ← function_body
  ← });
else if D==2 then
  ← parallel_for(blocked_range, [=](blocked_range2d<int> &range) {
    ← function_body
  ← });
else
  ← parallel_for(blocked_range, [=](blocked_range3d<int> &range) {
    ← function_body
  ← });
end if
```

9.5.3 Generated TBB code

The generated TBB code shown in Listing 9.5 shows the code generated for the update function using these patterns. This example shows a two-dimensional simulation using the Euler integration

method. A `blocked_range2d` structure is used to separate the lattice into 64x64 blocks which are processed in parallel using a `parallel_for` call.

Listing 9.5: TBB code automatically generated by STARGATES for a finite-differencing simulation using Euler integration in two-dimensions. Uses `parallel_for` to call `operator` for each compute block.

```
void euler(double u0, double u1, double h) {
  parallel_for(blocked_range2d<int>(0, Y, 64, 0, X, 64),
               [=](blocked_range2d<int> &range) {
    int yb = range.rows().begin();
    int ye = range.rows().end();
    for(int iy = yb; iy < ye; iy++) {
      int xb = range.cols().begin();
      int xe = range.cols().end();
      for(int ix = xb; ix < xe; ix++) {
        double u0yx = u0[iy*X + ix];
        //compute equation for cell ix,iy
        //u1 = u0 + f(u0) * h
      }
    }
  });
}
```

The only differences between the TBB implementation and the C implementation is the iteration and index calculation. For this reason no change to the main function is required. The main function for the TBB implementation will be the same as the C implementation main function in Listing 9.4.

9.6 Generating MPI code

The nature of an MPI implementation is different from a TBB or C implementation. The lattice is no longer stored in the memory of a single machine. Instead the lattice and program are distributed among multiple compute nodes. The MPI generator must produce code that performs necessary MPI functions such as determining the node's position within the lattice and communicating with neighbouring nodes. The code produced by this generator will be executed on each of the compute nodes that will collectively compute the simulation.

To generate the lattice allocation/initialisation and the MPI communication code, the generator must know what lattice decomposition method to use. This can be determined either by an in-built heuristic or as an additional input into the generator. It is up to the discretion of the generator designer to determine how to choose a lattice decomposition method.

The method will determine what size and shape lattice section each node should create as well as the code required to communicate border information to neighbouring nodes. As discussed in Chapter 5, each method will have different performance depending on the nature of the simulation, the size of the system and the number of compute nodes.

9.6.1 MPI Template

The MPI generator will use the template shown in Algorithm 12 to construct MPI implementations of simulations. The generator will create code for each node to identify itself and its neighbours, allocate memory, compute the simulation and communicate the necessary information to the neighbours.

Algorithm 12 The MPI generator template for constructing an MPI simulation. Generates a MPI program with a single update function that performs all the necessary steps of the integration method and communicates the bordering cells between each step.

```
generate includes

generate update function
for all stage in Steps do
  generate MPI iteration code
  generate neighbour access code
  for all equation in stage do
    traverse equation to generate equation code
  end for
  generate MPI communication code
end for

MAIN
generate MPI initialisation
generate parameter allocation
generate parameter initialisation
generate lattice allocation
generate lattice initialisation
generate time step iteration code
generate function call
generate end iteration code
```

The communication and iteration code will depend on the lattice decomposition method used. The following section presents the patterns to construct this code for a one-dimensional decomposition method as this was the method of decomposition presented in Chapter 5.

9.6.2 MPI Patterns

For the constructed simulation to make use of the MPI environment, the generator must produce code to initialise the MPI system and for the node to identify itself and its neighbouring nodes. The code given in Algorithm 13 will be used to perform this initialisation and identification.

The generator will produce code using a one-dimensional decomposition method so the lattice will be split in the highest dimension between `P` compute nodes. Thus the size of the lattices which each node will allocate will be **size**/`P` plus additional storage for the borders. The width of the borders Ĥalo is automatically determined by the generator. The equations in the simulation tree will be traversed to find the stencil with the largest width in the dimension of decomposition. The width of this stencil will determine the size of the borders and the value of `Halo`. The pattern for allocating lattices is shown in Algorithm 14.

Algorithm 13 MPI Generator - MPI initialisation.

```
← int id, P;
← MPI_Init(&argc, &argv);
← MPI_Comm_rank(MPI_COMM_WORLD, &id);
← MPI_Comm_size(MPI_COMM_WORLD, &P);
← idm1 = (id == 0) ? P-1 : id - 1;
← idp1 = (id == P-1) ? 0 : id + 1;
```

Algorithm 14 MPI Generator - lattice allocation pattern.

```
size = ((dim[D-1]/P) + 2*Halo)
for d = D-2..0 do
  size = size * dim[d]
end for
← type *name=(type*)malloc(size*sizeof(type));
```

When iterating over this kind of lattice, the bordering cells should not be updated. The borders will be read from and updated by the MPI communication, not from the iteration. Algorithm 15 shows the pattern for creating code to iterate over a lattice split in the highest dimension.

Algorithm 15 MPI Generator - lattice iteration pattern.

```
← for(int idim[d] = Halo; idim[d] <(dim[d]/P)+Halo; idim[d] ++) {
for d = D-1..0 do
  ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
end for
```

The code produced with this pattern will iterate over the entire lattice at once. For some communication methods it is advantageous to update the cells that must be sent to neighbouring nodes first (see Chapter 5). For a one-dimensional decomposition method this will result in three sets of iterations. These iteration loops can be produced with the pattern given in Algorithm 16.

Finally the pattern for the MPI communication calls is required. These patterns will produce communication calls for lattices split in one-dimension but others can easily be constructed for different decomposition methods. Borders of width `Halo` will be exchanged with the neighbouring nodes using `MPI_Isend/MPI_Irecv` calls. Algorithm 17 shows pattern for generating these calls.

This set of patterns can be used together or rearranged to implement different communication methods. If the borders are processed first, the communication can be initiated before the main iteration loop. Generating the `MPI_Waitall` function after this main loop will ensure the communication has completed before the next time step. This shows how the MPI generator can produce code with overlapping computation and communication.

9.6.3 Generated MPI Code

Listing 9.6 shows the relevant code for the generated MPI implementation. The main function contains code that initialises MPI and identifies the node's id and neighbours. The memory allocated for each node stores the section of the lattice for which the node is responsible for as well as the extra bor-

Algorithm 16 MPI Generator - lattice iteration pattern processing borders first.

```
← for(int idim[d] = Halo; idim[d] <2*Halo; idim[d] ++) {
for d = D-1..0 do
  ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
end for

← for(int idim[d] = (dim[d]/P); idim[d] <(dim[d]/P)+Halo; idim[d] ++) {
for d = D-1..0 do
  ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
end for

← for(int idim[d] = 2*Halo; idim[d] <(dim[d]/P); idim[d] ++) {
for d = D-1..0 do
  ← for(int idim[d] = 0; idim[d] <dim[d]; idim[d] ++) {
end for
```

Algorithm 17 MPI Generator - border communication pattern.

```
if D == 1 then
  pitch = 1
else
  pitch = dim[D-2]
  for d = D-3..0 do
    pitch = pitch*dim[d]
  end for
end if
← MPI_Request requests[4];
← MPI_Irecv(&name[(dim[D-1]/P+Halo)*pitch], Halo*pitch, type, idp1, 0,
            MPI_COMM_WORLD, &requests[0]);
← MPI_Irecv(&name[0],                       Halo*pitch, type, idm1, 0,
            MPI_COMM_WORLD, &requests[1]);
← MPI_Isend(&name[Halo*pitch],              Halo*pitch, type, idm1, 0,
            MPI_COMM_WORLD, &requests[2]);
← MPI_Isend(&name[dim[D-1]/P*pitch],        Halo*pitch, type, idp1, 0,
            MPI_COMM_WORLD, &requests[3]);
← MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
```

dering cells necessary to compute the simulation. In the update function each node will compute the equation for its section of the field and communicate the borders to its neighbours using `MPI_Isend` and `MPI_Irecv`. This code example processes the entire lattice at once and then communicates the data to the neighbours.

Listing 9.6: Generated MPI code for a two-dimensional finite differencing simulation using Euler integration. `euler` performs a single computation step for each node's field and communicates the borders to the neighbouring nodes.

```
void euler(float *u0,float *u1,float h) {
  for(int iy = Halo; iy < Y/P+Halo; iy++) {
    for(int ix = 0; ix < X; ix++) {
      u0yx = u0[iy*X + ix];
      //compute equation for cell ix,iy
      //u1 = u0 + f(u0) * h
    }
  }
  MPI_Irecv(&u1[((Y/P)+Halo)*X], Halo*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[0]);
  MPI_Irecv(&u1[0],              Halo*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[1]);
  MPI_Isend(&u1[Halo*X],         Halo*X, MPI_FLOAT, idm1, 0, MPI_COMM_WORLD, &requests[2]);
  MPI_Isend(&u1[Y/P*X],          Halo*X, MPI_FLOAT, idp1, 0, MPI_COMM_WORLD, &requests[3]);
  MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
}

int main(int argc, char **argv) {
  int id, P;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  idm1 = (id == 0) ? P-1 : id - 1;
  idp1 = (id == P-1) ? 0 : id + 1;
  ...
  float *u0 = new float[((Y/P)+2*Halo)*X];
  float *u1 = new float[((Y/P)+2*Halo)*X];
  for(int t = 0; t < 1024; t++) {
    euler(u0,u1,h);
    swap(&u0, &u1);
  }
  MPI_Finalize();
}
```

9.7 Generating CUDA code

The CUDA generator has another type of parallelism to consider when constructing simulations for the GPU. The generator must create code to allocate lattices in host memory as well as in the GPU device memory. CUDA calls to copy data between these two memory areas must also be created to copy the simulation in and out of the GPU. As there is no way to synchronise between threads in different blocks, multiple CUDA kernels must also be created to compute each stage of the chosen integration method.

9.7.1 CUDA Template

Algorithm 18 shows the high-level template the CUDA generator uses to construct simulation code. It should be noted that, unlike the previous templates, a different update function is generated for each integration stage. Different CUDA kernels are required to compute the different stages as it is the only way to synchronise all the threads in a CUDA application.

This generator uses patterns to implement CUDA simulations that use global memory. See Chapters 4 and 5 for more details on the CUDA memory types. This memory type was selected because it showed the highest performance on Fermi architecture GPUs.

Algorithm 18 Pseudo code for generating a CUDA finite-differencing solver. This generator creates one function for each integration step.

```
generate includes

for all stage in Steps do
  generate CUDA kernel
  generate thread id calculation
  generate neighbour access code
  for all equation in stage do
    traverse equation tree to generate equation code
  end for
end for

MAIN
generate CUDA initialisation
generate parameter allocation
generate parameter initialisation
generate lattice allocation
generate lattice initialisation
generate CUDA copy data from host to device
generate CUDA run-time parameters
generate time step iteration code
for all stage in Steps do
  generate CUDA call
end for
generate end iteration code
generate CUDA copy data from device to host
```

This template requires the use of several new patterns. The host allocation and iteration code will be the same as described in the C generator. But CUDA specific patterns are required to generate code for allocating device memory, copying data in and out of the device, calling kernel on the device, configuring the kernel calls and calculating an index on the device.

9.7.2 CUDA Patterns

To allocate data for the lattices, the generator must now allocate memory space on the host as well as the device memory. The generator can use the same pattern as the C generator to allocate lattice

memory on the host. To allocate memory on the device the generator must produce `cudaMalloc` calls. To avoid naming conflicts, this pattern inserts `d_` before the lattice name. The pattern for allocating a lattice on the GPU device is given in Algorithm 19.

Algorithm 19 CUDA Generator - lattice allocation pattern.

← **type** *d_**name**;
← cudaMalloc((void**) &d_**name**, **size***sizeof(**type**));

To copy data between the host and device, `cudaMemcpy` calls must be used. These calls represent the communication of data between the host and the device. Calls for each lattice will be generated at the start and at the end of the simulation, to copy the input into the device and the result back out. Two patterns are used to copy a lattice in and out of the device. These two patterns are shown in Algorithm 20.

Algorithm 20 CUDA Generator - lattice communication pattern.

← cudaMemcpy(d_**name**, **name**, **size***sizeof(**type**), cudaMemcpyHostToDevice);

← cudaMemcpy(**name**, d_**name**, **size***sizeof(**type**), cudaMemcpyDeviceToHost);

To construct a CUDA program, the code to construct the size of a block and a grid must be generated. The dimensionality of these constructs will depend on the dimensionality of the desired simulation. The following pattern can be used to create the code for these two structures with an array **grain** which specifies the size of the block in each dimension. Algorithm 21 shows the pattern to create the constructs.

Algorithm 21 CUDA Generator - lattice partition pattern.

size = dim[1]
for d = 2..D-1 **do**
  size = size * dim[d]
end for
← dim3 block(**grain[0]**, **grain[1]**, **grain[2]**);
← dim3 grid(**dim[0]**/**grain[0]**, **size**/(**grain[1]***grain[2]**));

CUDA threads created using these structures must calculate their unique index. Different calculations will be required to compute the index in each dimension. The following patterns in Algorithm 22 will construct code to calculate the indexes of the thread depending on the dimensionality of the simulation **D**.

The final pattern is the call that CUDA uses to launch a grid of threads to compute the update functions. As each stage of the integration must be computed by a different kernel, multiple kernel calls will be required. The code to call an update function **function_name** with parameters **parameters** can be be generated using the pattern in Algorithm 23.

Algorithm 22 CUDA Generator - index calculation pattern.

```
if D==1 then
  ← int idim[0] = (blockIdx.x*blockDim.x) + threadIdx.x;
else if D==2 then
  ← int idim[0] = (blockIdx.x*blockDim.x) + threadIdx.x;
  ← int idim[1] = (blockIdx.y*blockDim.y) + threadIdx.y;
else if D==3 then
  ← int idim[0] = (blockIdx.x*blockDim.x) + threadIdx.x;
  ← int idim[1] = ((blockIdx.y
  ← int idim[2] = ((blockIdx.y/(dim[1]/blockDim.y))*blockDim.z) + threadIdx.z;
else
  ← int k = (threadIdx.z*(gridDim.y*blockDim.y*gridDim.x*blockDim.x)) +
  (((blockIdx.y*blockDim.y) + threadIdx.y)*(gridDim.x*blockDim.x)) +
  (blockIdx.x*blockDim.x) + threadIdx.x;
  mod = 1
  div = 1
  for d = 0..D-1 do
    mod = mod * dim[d]
    ← int idim[d] = (k/div)%mod;
    div = div * dim[d]
  end for
end if
```

Algorithm 23 CUDA Generator - kernel call pattern.

```
← function_name <<<grid, block >>>(parameters);
← cudaThreadSynchronize();
```

9.7.3 Generated CUDA Code

The results of this code generator can be seen in Listing 9.7 which shows code elements for computing a two-dimensional simulation using the Euler integration method. The listing shows the code to allocate memory on the device, copy data in and out of the device, create structures for the block and grid, launch the update kernels and compute indexes within the device.

Listing 9.7: Generated two-dimensional Cahn-Hilliard simulation using Euler integration in CUDA. `block` and `grid` are parameters to control the threads to compute the simulation and `euler` is the function that each thread will perform.

```
__global__ void euler(float *un,float *unh,float h) {
  int ix = (blockIdx.x*blockDim.x) + threadIdx.x;
  int iy = (blockIdx.y*blockDim.y) + threadIdx.y;
  //compute equation for cell ix,iy
}
int main() {
  cudaSetDevice(0);
  ...
  float *d_u0, *d_u1;
  cudaMalloc((void**) &d_u0, X*Y*sizeof(float));
  cudaMalloc((void**) &d_u1, X*Y*sizeof(float));
  cudaMemcpy(d_u0, u0, X*Y*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_u1, u1, X*Y*sizeof(float), cudaMemcpyHostToDevice);
  dim3 block(32,8);
  dim3 grid(X/block.x, Y/block.y);
  for(int t = 0; t < 1024; t++) {
    euler<<< grid, block >>>(d_u0,d_u1,h);
    cudaThreadSynchronize();
    swap(&d_u0, &d_u1);
  }
  cudaMemcpy(u0, d_u0, X*Y*sizeof(float), cudaMemcpyDeviceToHost);
}
```

9.8 STARGATES Results

The measure of success for STARGATES is how well it can produce correct and efficient parallel code from simple equations descriptions. The generator has been tested with the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra computational models. Simulations have been generated in one-, two-, three- and four- dimensions using regular rectilinear lattices. The integration methods tested are the Euler, RK2, RK4 and DP6 methods. All the possible combinations of these elements can be correctly combined together and have been tested with the four generators discussed in this chapter to compute these simulations using - C, TBB, MPI and CUDA.

This is a total of 192 simulations that have been generated and tested. The performance of these generated simulations is indistinguishable from the hand-crafted versions. As the performance of these simulations is the same as the hand-written implementations, the performance results in Chapter 6 reflects the performance of the generated code. See Appendix B to see an example of a full simulation code produced by STARGATES.

9.9 Conclusions

This chapter has presented several algorithms for generating code from abstract simulation trees. Algorithm 3 showed how an equation tree can be recursively traversed to produce the code that computes the equation the tree represents. This algorithm can be coupled with the target specific generators to produce sequential or parallel code to compute the simulation.

These target specific generators have a lot of freedom in the way they are constructed which allows them full control over precision, parallelisation methods, communication etcetera. This allows parallel code to be generated for different architectures and languages that use varying methods of parallelisation. The advantage of this system is that these simulations can be created from the same simple abstract tree representations. The generated code presented in this chapter and Appendix B acts as proof that this generative programming system can construct simulation implementations for a number of different models, methods and target languages.

"I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tanhauser Gate. All those moments will be lost in time, like tears in rain."

Rutger Hauer in Ridley Scott's *Bladerunner*

10 Conclusions

10.1 Thesis Overview

This thesis has investigated issues in the study of computational model complexity through numerical simulation. The grounding theory of these computational models range from the approximation of natural systems to the completely abstract. These models can exhibit very similar behaviour, spatial patterns and growth regardless of their underlying theory. Due to the huge range of possible computational models, this research has been focused on one specific subset of models, namely field equations. Three field equations in particular have been discussed and used as examples throughout the thesis - they are the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations. These computational models are fully discussed in Chapter 2.

The behaviour and properties of these models can be studied and compared through computational simulation. Metrics of growth and spatial patterns can be developed to measure how the simulations evolve over time. Many systems initialised with different random seeds can be simulated to gather statistical data about the behaviour of the model. To compute these simulations, numerical methods must be used to transform the models to discrete approximations which can be stored and calculated by computer. The models considered in this work require discretisation in both space and time. There are a range of numerical methods which can be used to discretise the models. Finite-differencing and explicit Runge-Kutta integration methods have been chosen because of the ease with which they can be parallelised. These numerical methods and how they can be applied to computational models are discussed in Chapter 3.

To gather useful statistics for these models, many simulation runs and large system sizes may be required. Simulations of systems in a number of dimensions may also be necessary to identify dimension-specific behaviour. Computing this many simulations can be a computationally intensive task requiring the use of parallel computers. Chip manufacturers are reaching the physical limits of processor design which restricts the maximum computational throughput of a single core. Instead chip manufacturers are increasing the number of cores in a processor to continue to improve overall computational performance. There are many challenges which must be overcome to achieve this, mostly in the supporting architecture of the chip. Chapter 4 discusses and compares several shared-

memory and distributed-memory parallel computing architectures.

Implementing simulations on these parallel architectures requires the use of a parallel library or language. These languages/libraries allow simulations to be implemented with a number of different parallelisation strategies. Hybrid parallel architectures may require the use of multiple parallel languages/libraries to implement the simulations. Each of these languages/libraries have different advantages and disadvantages in terms of performance, flexibility, scalability and ease of programming. Several simulation codes implemented in these different languages must be maintained to allow the simulation to be executed on each of the different architectures. These parallel programming languages are discussed in Chapter 4 and the simulation implementations are discussed in Chapter 5. The performance of the different parallel architectures and implementations are compared in Chapter 6.

Writing and maintaining these different simulation implementations represents a large amount of work. New languages and libraries get released and existing ones are continually evolving. Maintaining an up-to-date simulation code base that makes full use of all available features is a tedious and time-consuming task. Generative programming has been presented as a viable alternative to maintaining hand-written code. Generative programming systems can be created to automatically produce target solutions within a specified program family. The desired elements are defined in a high-level Domain Specific Language and automatically combined by the system.

The program family identified in this research is simulations of field equations that can be numerically approximated using finite-differencing and explicit Runge-Kutta integration methods on rectilinear lattices. The elements that must be defined for a simulation are the governing equation(s), the finite-differencing stencils, the integration method and additional configuration information. A DSL that allows the user to define the elements of a desired simulation has been developed that allows a user to define these elements individually with simple ASCII representations. The DSL and examples are discussed in Chapter 7.

These element representations are parsed and stored internally as trees which allows them to be manipulated and combined appropriately. This high-level of information and knowledge about the program family allows the GP system to introduce parallelism to the simulations without the need for instruction level dependence analysis. All of the simulation elements are combined together into a single tree representing the entire simulation. This process is discussed in Chapter 8.

These simulation trees can be traversed and investigated by target specific code generators that produce the simulation implementations. These code generators have in-built templates for simulations which they use to construct simulation code. Generators can be written for any target language or library. Architecture and language specific optimisations can be introduced by the generators. The generators should analyse the simulation tree to determine if these optimisations can be used. Generator specific configuration can also be supplied to the target generators in the configuration file. Chapter 9 discusses these target generators and presents examples.

The developed GP system has been tested with three models - the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations. These models use different data types, finite-differencing stencils and equation/lattice combinations. The Euler, Runge-Kutta 2^{nd} order, Runge-Kutta 4^{th} order and Dormand-Prince 5^{th} order integration methods have been defined and tested with the system.

Code generators have been written which can successfully produce simulation implementations of these models for four different targets - C, TBB, MPI and CUDA.

10.2 Major Contributions

This thesis has shown that generative programming methods can successfully be applied to computational model simulations, reducing programmer effort but without loss of parallel performance. A Domain Specific Language can be created to allow various models and numerical methods of a simulation to be defined in a simple and easy-to-learn language. The process of parsing this language and automatically generating parallel simulations has been shown.

This system is currently limited to field-equation models that can be solved using finite differencing, explicit integration methods and regular rectilinear lattices. The system can easily be extended to encompass a wider range of computational models, numerical methods and lattice types without significant change to its structure. The generative programming system described in this system makes a number of important contributions.

Firstly it presents an alternative method of creating and maintaining a parallel simulation code base. The various simulation elements can be defined in a simple and concise way which are then combined automatically to produce a particular implementation. The implementations produced by this system can target any type of parallel architecture and use any parallel language or library. The implementations are also very lightweight which can overcome performance issues faced by more general simulation frameworks and libraries. It also avoids the duplication of work required in maintaining multiple code implementations.

The separation of a simulation definition from the implementation details also makes migrating simulations to new parallel architectures much easier. All existing simulations can be migrated to a new language or architecture by the development of a single new generator. This can be extremely beneficial for research groups that study many different simulations.

Secondly it provides interesting possibilities for non-expert developers to harness the power of parallel computing. Systems such as this can allow researchers to define simulations in forms and languages with which they are familiar. These simulation definitions can then be automatically turned into high-speed parallel implementations without the researchers requiring detailed knowledge of parallel architectures, languages or, in fact, parallelism in general.

Thirdly this type of system presents potential for researchers investigating and comparing the behaviour of computational models. By simplifying the definition of a simulation and significantly reducing implementation time, a great many more simulations can be implemented and compared by a single researcher.

Finally this type of system presents potential for a higher level of automatic parallelisation. The code generators currently contain templates for parallel implementations but this parallelism could be automated. As the dependence analysis would no longer depend on low-level instructions, the system could determine dependency at a much higher level and deduce a greater level of information. With the level of information about the simulation that can be deduced by the system, there is potential for parallelism to be automatically introduced by the system.

10.3 Implications for Automatic Parallelism

For over twenty years the speed of processors to compute sequential tasks has doubled in performance once every eighteen months. This performance increase has been driven by improvements in transistor manufacturing technology which allows the size of the processor pathways to be reduced and clock speeds to be increased in frequency. However, chip production has encountered a power wall which has stopped this increase in clock speed. A roadmap made in 2005 predicted that by 2010 clocks speeds should be over 15 GHz [97], in reality they are yet to reach 5GHz. Rather than continuing to pursue a faster single-core processor, most modern chip architectures include multiple cores into a single chip [94]. This hardware driven shift into parallelism presents software developers with a frightening realisation - the future of computing is parallel.

For decades parallel programming has had a firm placing in research labs [95] but has struggled to make the leap into industry. There are two major reasons for this - the ever-increasing speed of processors and the fact that parallel programming is hard. However, as the sequential processing speed of chips has stopped increasing, the industry must face the challenge of incorporating parallel programming into general use.

Parallel programming is hard because of the large potential for bugs. Programmers will have to address synchronisation, communication, data consistency, race conditions and deadlock in addition to normal programming considerations. One of the largest challenges with parallel programming is program verification. Methods such as unit testing are commonly used to assist development teams in producing correct software. Nowhere near the same level of supporting tools are available for parallel program development and the non-deterministic nature of parallel software makes such testing and support frameworks hard to implement.

Writing applications with explicit parallelism is hard to learn, even harder to master. Parallel languages are slowly becoming easier to use, but successfully teaching programmers to safely navigate the minefield of concurrency is a huge problem. However as the industry is forced to move into widespread parallel development it is a problem that must be addressed, and quickly. A large workforce of programmers, software developers and analysts must be retrained to think in parallel. If this retraining cannot be achieved, the progress of software software will stall dramatically as it will fail to make use of the hardware available.

A very attractive solution to this problem would be to move all parallelism into the compiler. Such compilers would allow software development methods to remain relatively unchanged and simply recompile existing code with parallel features automatically included. There have been many attempts over the years at creating compilers which automatically parallelise sequential code. These compilers have had limited success as they are restricted by the huge challenge of dependence analysis. For the moment, automatically parallelising compilers remain a holy grail of computer science.

Generative programming methods present an interesting alternative to automatically parallelising compilers. GP systems address program families rather than arbitrary sequential code. Because GP systems construct implementation from a high-level description of a desired solution, they have the potential to perform dependence analysis at this high level. This extra information and knowledge of the program family can allow a generative programming system to introduce parallelism at

any point in the assembly process.

One of the major limitations of generative programming systems is the complexity of constructing them. As a paradigm, generative programming methods are a relatively vague collection of templates, generators, analysis etc. This paradigm attained some popularity in the mid-1990s but seems to have fallen out of favour since then. With the looming parallel crisis, it may be time for generative programming to make a reappearance.

For generative programming systems to successfully influence the software industry, a number of issues must be addressed. While the individual components of a generative programming system are well-established, effective methods of combining them are not clearly defined. More general methods of defining program families, problem domains, combination rules and target generation will be required. Additional tools and possibly languages will be required to assist in the creation of generative programming systems. While a number of improvements in modern compiler technology can aid in this process, there is still significant gap.

The short-term advantage of generative programming system is the automation of parallelisation it can provide to developments teams which target a specific program family. The greater long-term benefits generative programming research could produce are improved methods for problem specification, high-level dependence analysis and automatic parallelisation.

Researching automatic parallelisation for specific program families rather than the huge task of parallelising arbitrary sequential programs could lead to important discoveries in dependence analysis and parallelisation. These discoveries have the potential to lead to major improvements in automatic parallelisation and incrementally work towards arbitrary automatically parallelising compilers.

10.4 Implications for Computational Simulations

Computational simulations encompass a wide range of scientific and industrial applications. There exist a number of simulation paradigms including agent models, cellular automata, field approximations, network models and particle models. Simulations within these paradigms can be used to approximate a huge range of real-world systems. Many systems require the combination of multiple simulation paradigms to approximate them. These computational simulations have applications in the study of physics, chemistry, biology, social systems etcetera.

The generative programming system described in this work has interesting implications for computational simulations. The simulations produced by the system are limited to a specific class of field equations. It allows the elements of this type of simulation to be defined in a high-level and simple way and automatically constructs fast parallel code.

The success of this system suggests that it would be possible to construct similar systems for the other simulation paradigms. Such systems would have immediate benefits for computational simulation researchers. The ease with which simulations can be modified and deployed to parallel implementations is the obvious advantage. It also provides benefits in terms of maintainability and migration to new processing architectures.

In the long term this could lead to significant developments in simulation definition languages.

Creating domain specific languages for the different simulation families would lead to a better understanding of simulation elements and the ways of describing them. This could lead to a merging of languages into a single simulation language.

A language capable of describing simulations within the entire range of paradigms would be a powerful tool in a simulation designer's workshop. These general simulation descriptions could lead to a generative programming system that constructs implementations for a huge range of simulations. Such a system coupled with the potential for high-level dependence analysis and automatic parallelisation would be capable of automatically producing parallel simulations.

A system such as this would not be an arbitrary automatically parallelsing compiler as it would be limited to computational simulations. However, it would be a significant leap forward in parallelism and automation. The construction of this system would require a large amount of effort over many years. However, it would be an extremely powerful tool which could enable researchers to investigate more complex simulations that are currently infeasible to construct.

10.5 Future Work

There are a number of ways in which this research can be extended. It could be achieved by adding support for additional computational models, numerical methods and parallel architectures. These additions should not require any significant changes to the existing structure of the system.

It is important to ensure that any additional features do not include simulations that fall outside of the program family. The strength of generative programming systems is their focus towards producing solutions for a specific program family. This focus allows the system to automate the process of combining elements and producing solutions. Extending the system to include simulations outside of the program family may compromise the validity of the system.

10.5.1 Computational Models

One obvious way to improve this system would be to extend it to allow a wider range of computational models. Field equations encompass a large number of models and automatically generating simulations of them can significantly decrease the amount of work required to compare different models. These field equations can be based in Mathematics, Physics, Chemistry, Biology or the completely abstract. However, it would be even more useful to extend the equation parsing and equation trees to include cellular automata.

These models (such as the Ising model [154] or Lattice Gas model [155]) are different in nature to the class of field equations addressed in this thesis. Such models are often governed by logical expressions rather than equations and also often include some random components. These models would require the inclusion of logical statements to the equation grammar and the modification of the generators to be able to produce code to compute these models.

Another change that would be required to include such models would be a change to the update models. These abstract models often do not use the same type of integration methods as field equations. Abstract models often do not need to be integrated over time but are simply updated with a

single stage update. Some complexities arise with such updates and the models may require update methods such as the checkerboard update to avoid parallel issues or sweeping effects.

Before including such methods into STARGATES it must first be determined if they fall within the same program family as field equations. Generative Programming methods are designed to produce solutions within a single program family. Attempting to force a Generative Programming system to produce solutions of two different program families could result in an inflexible system. It may be necessary to create a separate GP system to produce simulations of this type of computational model.

10.5.2 Integration Methods

Adaptive stepsize integration methods would be one useful extension for STARGATES. These methods could be described to STARGATES by allowing the inclusion of a b' line on the Butcher tableau describing the method. They are controlled by a parameter e which limits the maximum acceptable error. It would be trivial for this parameter to replace the time step parameter h used by the fixed time step methods.

For the simulations to utilise an adaptive stepsize method, a process for determining the error of the entire system must be determined. If a common process for computing this can be developed then it would be relatively simple to include adaptive methods. The Butcher tableau parser can simply check if a b' line exists or not. These adaptive stepsize methods are discussed further in Appendix A.

The system could be improved with the inclusion of implicit integration methods. Implicit Runge-Kutta integration methods can also be described by Butcher tableau but these tableaux will no longer be restricted to lower-triangular. Like the adaptive stepsize methods, whether an integration method is explicit or implicit can be determined by examining the tableau.

These implicit integration methods are harder to parallelise as they require methods for solving a set of equations. This type of integration method can also be controlled by a parameter e which controls the maximum allowable error in the solution. These solving methods are often hard to parallelise but, if a common method can be identified, then such methods can be included into the system. These implicit methods can integrate some simulations which simply cannot be integrated explicitly.

10.5.3 Numerical Methods

Another useful extension to the system would be the inclusion of finite elements. Solutions to the class of models discussed in this thesis can be approximated using finite differencing, however this is not true for all computational models. Finite element methods are required to approximate solutions to many different types of model.

Including support for finite element methods would greatly extend the applicability of the system but would represent a significant amount of work. The commonalities of finite element simulations must be identified, a method of describing them must be developed and a regular process for parallelising them must be determined. Inclusion of these methods would require significant changes to the system and a considerable amount of work.

If finite element methods are included into the system, it must be in such a way that the performance of the produced finite different solutions is not affected. Correctly designed code generators should be capable of producing both finite difference and finite element solutions.

10.5.4 Lattice Structures

The simulations considered in this thesis use one or more regular rectilinear lattices to approximate the systems. There are many other possible lattice structures such as hexagonal or triangular structures; the lattices can also have irregular structures. Some models require these alternate lattice structures to function correctly.

One example of this is the Lattice Gas model [155] which simply cannot operate correctly on a rectilinear lattice. The symmetry of a hexagonal grid is required for the model to show the desired behaviour.

Another possibility for lattice structures is staggered grids. This structure is useful for systems that are approximated by one or more lattices. Rather than having the lattices centred around the same coordinates, the lattices are offset from each other. This offset is often one half of the cell width in each dimension. This can also be achieved by creating different stencils to access the different lattices, but this is not an elegant or easy to understand solution.

Theses additional lattice structures would also require changes to the numerical methods used to approximate solutions to the models. The different lattices will have different stencils to approximate the spatial operators. Simulating models on alternate lattice structures may require the use of finite elements as opposed to finite differencing. As discussed in the previous section this would require a significant amount of work.

10.5.5 Boundary Conditions

An improved method for defining boundary conditions would be extremely beneficial to this generative programming system. Currently the user can select between Periodic, Dirichlet or Neumann boundary conditions in each dimension. While these conditions are sufficient for the simulations considered in this work, they will not be suitable for a wider range of computational simulations.

A higher level of describing and constructing boundary conditions would be more appropriate. Developing a method of defining arbitrary boundary conditions would require a more extensive investigation of boundary conditions to determine their differences and commonalities.

This would also require adjustments to the finite difference (and possibly finite element) components of the system. The stencils used to compute the model equations must be kept consistent with the boundary conditions.

10.5.6 Parallel Generators

The easiest and most common method of extending the system is the modification of existing code generators and the inclusion of new ones. These generators can be modified and added with no impact on the rest of the system. Generators can be modified to include additional heuristics that

automatically tune the optimal parameters for the parallel update methods and include additional optimisations. As the languages of these generators change and improve, additional language features may be exploited by the generators to achieve the best performance possible.

Generators can also be written to utilise new parallel architectures, languages or libraries. These new generators allow all the simulations to be easily migrated to any new parallel architecture released in future years. Existing generators can also be improved to make use of improved target specific update methods and algorithms.

Generators that produce code for different languages (parallel or not) would also be useful. This would allow the system to compare different languages by implementing a number of simulations for different targets. Possible target languages for these generators include - OpenCL, OpenMP, High Performance Fortran, Java, Python or Fortran. In summary this approach seems to hold great promise for the future of complex system simulations.

"I go, and it is done; the bell invites me.
Hear it not, Duncan, for it is a knell
That summons thee to heaven or to hell."

A

Adaptive Stepsize Methods

A.1 Introduction to Adaptive Stepsize Methods

As the complexity of the solution of the models will not be constant over the entire simulation, a fixed time step can often waste valuable computation time. Integration methods that use adaptive time steps can be developed to adjust the time step used according to an approximate error calculation. These error approximations usually take the form of computing two solutions of different accuracy and taking the difference between them as an approximation of the error. This can be performed as either two completely separate steps or a more efficient approach is to use a method that computes to solutions using the same intermediate stages.

A.2 Runge-Kutta Methods with Adaptive Stepsizes

The methods discussed so far all approximate solutions with fixed time steps. The error these Runge-Kutta methods introduce into the system depends on the nature of the system and the time step of the method. In many cases the complexity of the solution is not constant throughout the entire simulation. The amount of error introduced by these methods varies over time and thus a fixed time step can often be wasteful. To overcome this waste, adaptive stepsizes can be used to adjust the length of the time step based on the complexity of the system. To adjust the time step, some indication of error must be calculated to know if a time step is too long and introducing too much error or unnecessarily short and wasting cycles.

One problem to be addressed is how to control the stepsize over an entire lattice. Because the complexity of the system will not be equal across the entire lattice, the error introduced by the integration method will not be equal either. One solution to this problem is to use the maximum error value of the entire lattice to control the stepsize. This will result in many parts of the lattice being integrated with an unnecessarily small stepsize but is chosen as it is easier to parallelise. Because each cell in the lattice is coupled to the neighbouring cells by the spatial terms of the system's governing equation, integrating neighbouring cells with different stepsizes presents a significant challenge that will have a large impact on parallel performance. For this reason it is not considered in this work.

One common way of controlling the step-size is the Richardson extrapolation method [62]. This method compares two solutions computed with a step-size of h and $\frac{h}{2}$. These two solutions can then be compared to provide an estimate of the error of the solution. However, this is an expensive method of estimating the error as two separate solutions must be computed.

Runge-Kutta methods have been developed which use the same intermediate values to compute two solutions with different orders of accuracy. The difference between these two solutions can be used to give an indication of the error of the lower order solution and be used to control the step-size of the method. The first such method appears to have been developed by Merson in 1957 [62, 156]. However, improved methods have subsequently been developed by Fehlberg [87], Verner [88] and Dormand-Prince [90].

A.2.1 Merson Method

The Merson method of integration uses two sets of coefficients to compute two solutions with different orders of accuracy. These two solutions can then be compared to give an estimate of the error of the solution. The advantage of these methods over the Richardson extrapolation method is that the same intermediate values are used to compute the two solutions. These two solutions are computing using two different sets of coefficients - b, which is used to compute the lower-order solution, and b', which is used to compute the higher-order solution. This method is shown in Table A.1.

Table A.1: Butcher Tableau representation of the Merson method.

0					
$\frac{1}{3}$	$\frac{1}{3}$				
$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$			
$\frac{1}{2}$	$\frac{1}{8}$	0	$\frac{3}{8}$		
1	$\frac{1}{2}$	0	$-\frac{3}{2}$	2	
b	$\frac{1}{6}$	0	0	$\frac{2}{3}$	$\frac{1}{6}$
b'	$\frac{1}{10}$	0	$\frac{3}{10}$	$\frac{2}{5}$	$\frac{1}{5}$
e	$-\frac{1}{15}$	0	$\frac{3}{10}$	$-\frac{4}{15}$	$\frac{1}{30}$

The difference between these two solutions can be used to give an estimate of the error. This can be computed explicitly using the row of coefficients given by e. This notation is used for all other methods with adaptive step-sizes.

A.2.2 Fehlberg Methods

Erwin Fehlberg describes Runge-Kutta methods with step-size control for many different orders of accuracy [62, 87, 157]. Fehlberg presents a number of these methods from first- to eighth-order accuracy. Like the Merson methods two solutions are computed with coefficients b and b' while e provides an estimate of the error. The $5^{th}/6^{th}$ order accurate method is given in Table A.2.

Table A.2: Butcher Tableau representation of the Fehlberg $5^{th}/6^{th}$ integration method. The b row of coefficients give a 5^{th} order accurate solution while the b' gives 6^{th} order accuracy. The difference between these two solutions can be used as an estimation of the error as calculated with the coefficients e.

0								
$\frac{1}{6}$	$\frac{1}{6}$							
$\frac{4}{15}$	$\frac{4}{75}$	$\frac{16}{75}$						
$\frac{2}{3}$	$\frac{5}{6}$	$-\frac{8}{3}$	$\frac{5}{2}$					
$\frac{4}{5}$	$-\frac{8}{5}$	$\frac{144}{25}$	-4	$\frac{16}{25}$				
1	$\frac{361}{320}$	$-\frac{18}{5}$	$\frac{407}{128}$	$-\frac{11}{80}$	$\frac{55}{128}$			
0	$-\frac{11}{640}$	0	$\frac{11}{256}$	$-\frac{11}{160}$	$\frac{11}{256}$	0		
1	$\frac{93}{640}$	$-\frac{18}{5}$	$\frac{803}{256}$	$-\frac{11}{160}$	$\frac{99}{256}$	0	1	
b	$\frac{31}{384}$	0	$\frac{1125}{2816}$	$\frac{9}{32}$	$\frac{125}{768}$	$\frac{5}{66}$	0	0
b'	$\frac{7}{1408}$	0	$\frac{1125}{2816}$	$\frac{9}{32}$	$\frac{125}{768}$	0	$\frac{5}{66}$	$\frac{5}{66}$
e	$-\frac{5}{66}$	0	0	0	0	$-\frac{5}{66}$	$\frac{5}{66}$	$\frac{5}{66}$

Fehlberg's methods are designed to use the solution provided by b and use the higher order solution b' to estimate the error. However, in practice it is often the higher order solution provided by b' that is propagated. This will cause the estimation of error to be no longer strictly correct [62] but is suitable for many practical applications.

This is one of the lower order integration methods described by Fehlberg. Runge-Kutta methods with step-size control of up to $8^{th}/9^{th}$ order accuracy are described [87]. These methods require many steps and result in a large number of coefficients.

A.2.3 Verner Methods

The Fehlberg methods have some identical values for the b and b' coefficients which can lead to some over-optimistic estimations of error in certain circumstances [62]. The methods developed by Verner overcome this issue by using non-identical coefficients. A $5^{th}/6^{th}$ order Verner method is shown in Table A.3.

Like the Fehlberg methods it is often the higher order b' solution that is used as the solution of the method. Again this leads to an inaccurate estimation of error provided by e but does provide a more accurate solution.

A.2.4 Dormand-Prince Methods

Given that the higher order accuracy solution is the one that is normally propagated, it makes sense that the method should be designed with this in mind. This is the approach taken by Dormand and Prince in developing their methods [62,89]. Dormand and Prince describe a number of these methods

Table A.3: Butcher Tableau representation of the Verner $5^{th}/6^{th}$ integration method.

0								
$\frac{1}{18}$	$\frac{1}{18}$							
$\frac{1}{6}$	$-\frac{1}{12}$	$\frac{1}{4}$						
$\frac{2}{9}$	$-\frac{2}{81}$	$\frac{4}{27}$	$\frac{8}{81}$					
$\frac{2}{3}$	$\frac{40}{33}$	$-\frac{4}{11}$	$-\frac{56}{11}$	$\frac{54}{11}$				
1	$-\frac{389}{73}$	$\frac{72}{73}$	$\frac{5380}{219}$	$-\frac{12285}{584}$	$\frac{2695}{1752}$			
$\frac{8}{9}$	$-\frac{8716}{891}$	$\frac{656}{297}$	$\frac{39520}{891}$	$-\frac{416}{11}$	$\frac{52}{27}$	0		
1	$\frac{3015}{256}$	$-\frac{9}{4}$	$-\frac{4219}{78}$	$\frac{5985}{128}$	$-\frac{539}{384}$	0	$\frac{693}{3328}$	
b	$\frac{3}{80}$	0	$\frac{4}{25}$	$\frac{243}{1120}$	$\frac{77}{160}$	$\frac{73}{700}$	0	0
b'	$\frac{57}{640}$	0	$-\frac{16}{65}$	$\frac{1377}{2240}$	$\frac{121}{320}$	0	$\frac{891}{8320}$	$\frac{2}{35}$
e	$\frac{33}{640}$	0	$-\frac{132}{325}$	$\frac{891}{2240}$	$-\frac{33}{320}$	$-\frac{73}{700}$	$\frac{891}{8320}$	$\frac{2}{35}$

in a whole range of orders of accuracy. A full discussion of how these methods are constructed can be found in [62, 89, 90]. The tableau of the commonly used $4^{th}/5^{th}$ order Dormand-Prince method can be found in Table A.4.

Table A.4: Butcher Tableau representation of the Dormand-Prince $4^{th}/5^{th}$ order integration method.

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{335}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
b	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
b'	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$
e	$-\frac{71}{57600}$	0	$\frac{71}{16695}$	$-\frac{71}{1920}$	$\frac{17253}{339200}$	$-\frac{22}{525}$	$\frac{1}{40}$

B

Generated Code Example

This appendix contains an example simulation constructed by the STARGATES system. This simulation code is presented in its entirety to show the complete code rather than the code fragments discussed throughout the rest of the thesis. This simulation is implemented in C and will compute the Cahn-Hilliard equation in two dimensions using the Euler integration method. This simulation is shown in Listing B.1.

Listing B.1: Generated simulation of the Cahn-Hilliard equation in two dimensions using the Euler integration method.

```
#include<stdio.h>

int X = 1024;
int Y = 1024;
float K = 1.0;
float U = 1.0;
float M = 1.0;
float B = 1.0;
void euler(float *u0,float *u1,float h) {
   for(int iy = 0; iy < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         int xm2 = ix-2;
         int xm1 = ix-1;
         int xp1 = ix+1;
         int xp2 = ix+2;
         int ym2 = iy-2;
         int ym1 = iy-1;
         int yp1 = iy+1;
         int yp2 = iy+2;
         xm2 = (xm2 < 0)  ? (xm2 + X) : (xm2);
         xm1 = (xm1 < 0)  ? (xm1 + X) : (xm1);
         xp1 = (xp1 >= X) ? (xp1 - X) : (xp1);
         xp2 = (xp2 >= X) ? (xp2 - X) : (xp2);
         ym2 = (ym2 < 0)  ? (ym2 + Y) : (ym2);
         ym1 = (ym1 < 0)  ? (ym1 + Y) : (ym1);
         yp1 = (yp1 >= Y) ? (yp1 - Y) : (yp1);
         yp2 = (yp2 >= Y) ? (yp2 - Y) : (yp2);
         float u0ym2x   = u0[ym2*X + ix ];
         float u0ym1xm1 = u0[ym1*X + xm1];
         float u0ym1x   = u0[ym1*X + ix ];
         float u0ym1xp1 = u0[ym1*X + xp1];
```

```
            float u0yxm2   = u0[iy*X  + xm2];
            float u0yxm1   = u0[iy*X  + xm1];
            float u0yx     = u0[iy*X  + ix ];
            float u0yxp1   = u0[iy*X  + xp1];
            float u0yxp2   = u0[iy*X  + xp2];
            float u0yp1xm1 = u0[yp1*X + xm1];
            float u0yp1x   = u0[yp1*X + ix ];
            float u0yp1xp1 = u0[yp1*X + xp1];
            float u0yp2x   = u0[yp2*X + ix ];
            u1[iy*X + ix] = (u0[iy*X + ix]+
            (((M*((((-B)*( (u0ym1x) + (u0yxm1) + (-4*u0yx) + (u0yxp1) + (u0yp1x)))
            +(U*( ((u0ym1x*u0ym1x)*u0ym1x) +
            ((u0yxm1*u0yxm1)*u0yxm1) + (-4*(u0yx*u0yx)*u0yx) + ((u0yxp1*u0yxp1)*u0yxp1) +
            ((u0yp1x*u0yp1x)*u0yp1x))))
            -(K*( (u0ym2x) +
            (2*u0ym1xm1) + (-8*u0ym1x) + (2*u0ym1xp1) +
            (u0yxm2) + (-8*u0yxm1) + (20*u0yx) + (-8*u0yxp1) + (u0yxp2) +
            (2*u0yp1xm1) + (-8*u0yp1x) + (2*u0yp1xp1) +
            (u0yp2x))))))*h));
         }
      }
   }

int main() {
   float *u0 = (float*) malloc(Y*X*sizeof(float));
   float *u1 = (float*) malloc(Y*X*sizeof(float));
   for(int iy = 0; iy < Y; iy++) {
      for(int ix = 0; ix < X; ix++) {
         u0[iy*X + ix] = (((rand()/(float)RAND_MAX)*2.0)-1.0);
      }
   }
   float h = 0.01;
   for(int t = 0; t < 1024; t++) {
      euler(u0,u1,h);
      float *ut = u0;
      u0 = u1;
      u1 = ut;
   }
}
```

Bibliography

[1] Wolfram S. *A New Kind of Science*. Wolfram Media, 2002.

[2] Bossomaier TRJ. *Complex Systems*. Cambri, 2000.

[3] Eames I, Flor JB. New Developments in Understanding Interfacial Process in Turbulent Flows. *Philosophical Transactions of The Royal Society* 2011; **369**:702–705.

[4] Cahn JW, Hilliard JE. Free Energy of a Nonuniform System. I. Interfacial Free Energy. *The Journal of Chemical Physics* 1958; **28**(2):258–267.

[5] Hawick KA. Domain Growth in Alloys. PhD Thesis, Edinburgh University 1991.

[6] Maxwell JC. *A Treatise on Electricity and Magnetism*. Clarendon Press, 1873.

[7] Watson RE, Blume M, Vineyard GH. Classical Heisenberg Magnet in Two Dimensions. *Physical Review B* 1970; **2**(3):684–690.

[8] Peczak P, Ferrenberg AM, Landau DP. High-accuracy Monte Carlo study of the three-dimensional classical Heisenberg ferromagnet. *Physical Review B* March 1991; **43**(7):6087–6093.

[9] Batchelor GK. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.

[10] Simon HD ((ed.)). *Parallel Computational Fluid Dynamics*. Scientific & Engineering Computation, MIT Press, 1992.

[11] Abbott MB, Basco DR. *Computational Fluid Dynamics: An Introduction for Engineers*. Longman, 1989.

[12] Newton I. *Philosophiæ Naturalis Principia Mathematica*. apud Sa. Smith: London, 1687.

[13] Schrödinger E. An Undulatory Theory of the Mechanics of Atoms and Molecules. *Physical Review* December 1926; **28**(6):1049–1070.

[14] Einstein A. Die Feldgleichungen der Gravitation. *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften* 1915; :844–847.

[15] Einstein A. Die Grundlage der allgemeinen Relativitätstheorie. *Annalen der Physik* 1916; **49**:769–822.

[16] Lotka AJ. *Elements of Physical Biology*. Williams and Wilkings Company, 1925.

[17] Volterra V. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Mem. R. Accademia Nazionale dei Lincei* 1926; **2**:31–113.

[18] Ginzburg VL. On Superconductivity and Superfluidity. *Les Prix Nobel - The Nobel Prizes 2003*. Nobel Foundation, 2004; 96–127.

[19] Ginzburg VL, Landau LD. On the Theory of Superconductivity. *Zhurnal Eksperimental'noi i Teoreticheskoi Fiziki* 1950; **20**:1064–1082.

[20] Cannon JR. *The One-Dimensional Heat Equation*. Encyclopedia of Mathematics and its Applications, Addison-Wesley Publishing Company, 1984.

[21] Vu VT, Cats G, Wolters L. Asynchronous Communication in the HIRLAM Weather Forecast Model. *Proc. 14th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*, 2008.

[22] Lynch P. The origins of computer weather prediction and climate modeling. *Journal of Computational Physics* March 2008; **227**:3431–3444.

[23] Fox GC, Williams RD, Messina PC. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994.

[24] Anderson JP, Hoffman SA, Shifman J, Williams RJ. D825 - A Multiple-Computer System for Command & Control. *Proceedings of the December 4-6, 1962, fall joint computer conference*, AFIPS '62 (Fall), ACM: New York, USA, 1962; 86–96.

[25] Barnes GH, Brown RM, Kato M, Kuck DJ, Slotnick DL, Stokes RA. The ILLIAC IV Computer. *IEEE Transactions on Computers* August 1968; **17**:746–757.

[26] Reddaway SF. DAP a Distributed Array Processor. *Proceedings of the 1st Annual Symposium on Computer Architecture*, ACM Press: Gainesville, Florida, 1973; 61–65.

[27] Batcher KE. *Algorithmically Specialized Parallel Computers*, chap. MPP: A high speed image processor. Academic Press, 1985.

[28] Lakshmivarahan S ((ed.)). *Proceedings of the Workshop on Parallel Processing using the Heterogeneous Element Processor*, 1985.

[29] Cray Research Inc. The Cray X-MP Series of Computer Systems.

[30] TOP500org. TOP 500 Supercomputer Sites. `http://www.top500.org/`. Last accessed June 2011.

[31] NVIDIA® Corporation. *CUDA™ 2.0 Programming Guide* 2008. Last accessed November 2008.

[32] Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.* 2004; **23**(3):777–786.

[33] McCool M, Toit SD. *Metaprogramming GPUs with Sh*. A K Peters, Ltd., 2004.

[34] Blume W, Eigenmann R, Hoe J, Padua D, Petersen P, Rauchwerger L, Tu P. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology* 1994; **2**.

[35] Schulte W, Tillmann N. Automatic Parallelization of Programming Languages: Past, Present and Future. *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, ACM: New York, USA, 2010; 1–1.

[36] Burke MG, Cytron RK. Interprocedural Dependence Analysis and Parallelization. *SIGPLAN Notices* April 2004; **39**:139–154.

[37] Kennedy K, Koelbel C, Zima H. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ACM: New York, USA, 2007; 7–1–7–22.

[38] High Performance Fortran Forum. *High Performance Fortran Language Specification*.

[39] Fox G, Hiranandani S, Kennedy K, Koelbel C, Kremer U, Tseng CW, Wu MY. Fortran D Language Specification. *Technical Report* 1990.

[40] Zima H, Brezany P, Chapman B, Mehrotra P, Schwald A. Vienna Fortran - A Language Specification Version 1.1. *Technical report*, Austrian Center for Parallel Computation March 1992.

[41] Thinking Machines Corporation. *CM Fortran Programming Guide*. 1.0 edn. January 1991.

[42] Czarnecki K, Eisenecker U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley: Boston, MA, 2000.

[43] Consel C. From a Program Family to a Domain-Specific Language. *Domain-Specific Program Generation*, *Lecture Notes in Computer Science*, vol. 3016, Lengauer C, Batory D, Consel C, Odersky M (eds.). Springer Berlin / Heidelberg, 2004; 37–291.

[44] Batory D. The Road to Utopia: A Future for Generative Programming. *Domain-Specific Program Generation*, *Lecture Notes in Computer Science*, vol. 3016, Lengauer C, Batory D, Consel C, Odersky M (eds.). Springer Berlin / Heidelberg, 2004; 211–250.

[45] Czarnecki K, Eisenecker UW. Components and Generative Programming. *SIGSOFT Softw. Eng. Notes* October 1999; **24**:2–19.

[46] Kaneko K, Tsuda I. *Complex Systems: Chaos and Beyond*. Springer, 2000.

[47] Stocker R, Jelinck H, Burnota B, Bossomaier T ((eds.)). *Complex Systems: From Local Interactions to Global Phenomena*. IOS Press, 1996.

[48] Allen M, Tildesley D. *Computer simulation of liquids*. Clarendon Press, 1987.

[49] Gould H, Tobochnik J, Christian W. *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. 3rd edn., Addison Wesley, 2006.

[50] Hockney RW, Eastwood JW. *Computer Simulation Using Particles*. Taylor & Francis, 1989.

[51] Fijany A, Jensent AM, Rahmat-samiit Y, Barhen J. A Massively Parallel Computation Strategy for Time and Space Parallelism Applied to Electromagnetic Problems. *IEEE Trans. on Antenna and Propagation* 1995; **43**:1441–1449.

[52] Adams S, Payne J, Boppana R. Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors. *HPCMP Users Group Conference* 2007; **0**:334–338.

[53] Kim KH, Kim K, Park QH. Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. *Computer Physics Communications* 2011; **182**(6):1201 – 1207.

[54] Jacobsen DA, Thibault JC, Senocak I. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. *48th AIAA Aerospace Sciences Meeting and Exhibit*, American Institute of Aeronautics and Astronautics (AIAA), 2010.

[55] Mineter M, Hulton N. Parallel Processing for Finite-Difference Modelling of Ice-Sheets. *Computers & Geosciences* 2001; **27**:829–838.

[56] Micikevicius P. 3D Finite Difference Computation on GPUs using CUDA. *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM: New York, NY, USA, 2009; 79–84.

[57] Guiffaut C, Mahdjoubi K. A parallel FDTD algorithm using the MPI library. *Antennas and Propagation Magazine, IEEE* April 2001; **43**(2):94 –103.

[58] Yang D, Liao C, Jen L, Xiong J. A parallel FDTD algorithm based on domain decomposition method using the MPI library. *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, 2003; 730–733.

[59] Rodohan D, Saunders S. Parallel implementations of the Finite Difference Time Domain (FDTD) method. *Computation in Electromagnetics, 1994. Second International Conference on*, 1994; 367–370.

[60] Runge C. Über die numerische Auflösung von Differentialgleichungen. *Mathematische Annalen* 1895; **46**:167–178.

[61] Kutta MW. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschrift für Mathematick und Physik* 1901; **46**:135–453.

[62] Butcher JC. *Numerical Methods for Ordinary Differential Equations*. J. Wiley, 2008.

[63] Dongarra JJ, Croz JD, Duff IS, Hammarling S. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 1990; **16**:18–28.

[64] Dongarra JJ, Moler CB, Bunch JR, Stewart GW. *LINPACK Users' Guide*. SIAM, 1979.

[65] Dongarra JJ. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[66] Bischof CH, Dongarra JJ. A Linear Algebra Library for High-Performance Computers. *Parallel Supercomputing: Methods, Algorithms and Applications*, Carey GF (ed.). chap. 4, Wiley, 1989; 45–55.

[67] Czarnecki K. Generative Programming - Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD Thesis, Technical University of Ilmenau 1998.

[68] Garcia S, Lopez R, Pantoja M, Bretones A. How to Create Complex FDTD FORTRAN and C Codes Simply with MATHEMATICA. *Antennas and Propagation Magazine, IEEE* June 2007; **49**(3):59–67.

[69] van Engelen R, Wolters L, Cats G. Automatic Code Generation for High Performance Computing in Environmental Modeling 1996.

[70] van Engelen R, Wolters L, Cats G. CTADEL: A Generator of Efficient Code for PDE-based Scientific Applications 1995.

[71] Akers R, Kant E, Randall C, Steinberg S, Young R. SciNapse: a problem-solving environment for partial differential equations. *Computational Science Engineering, IEEE* July-September 1997; **4**(3):32–42.

[72] van Engelen R, Wolters L, Gerard Cats G. CTADEL: a generator of multi-platform high performance codes for PDE-based scientific applications. *Proceedings of the 10th international conference on Supercomputing*, ICS '96, ACM: New York, NY, USA, 1996; 86–93.

[73] Logg A, Wells GN. DOLFIN: Automated Finite Element Computing. *ACM Trans. Math. Soft.* April 2010; **37**(2):1–28.

[74] Logg A. Automating the Finite Element Method. *Arch. Comput. Methods Eng.* 2007; **14**(2):93–138.

[75] Hawick KA, Playne DP. Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation. *International Journal of Computer Aided Engineering and Technology (IJCAET)* 2010; **2**:78–93.

[76] Carcke H, Niethammer B, Rumpf M, Weikard U. Transient coarsening behaviour in the Cahn-Hilliard model. *Acta Materialia* 2003; **51**:2823–2830.

[77] Bardeen J, Cooper LN, Schrieffer JR. Theory of Superconductivity. *Physical Review* December 1957; **108**(5):1175–1204.

[78] Hawick KA, Playne DP. Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA. *Proceedings IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2010.

[79] van Saarloos W. The Complex Ginzburg-Landau Equation for Beginners. *Spatiotemporal Patterns in Nonequilibrium Systems*, Cladis PE, Palffy-Muhoray P (eds.). Addison-Wesley, 1994.

[80] Abrikosov AA. On the Magnetic Properties of Superconductors of the Second Group. *Sov. Phys. JETP* 1957; **5**:1174.

[81] Abrikosov AA. Type II Superconductors and the Vortex Lattice. *Les Prix Nobel - The Nobel Prizes 2003*, Frangsmyr T (ed.). Nobel Foundation, 2004; 59–67.

[82] Aranson I, Kramer L. The world of the complex Ginzburg-Landau equation. *Rev. Mod. Phys.* 2002; **74**:99–143.

[83] Standish RK. Cellular Ecolab. *Complexity International* 1998; **6**:1–12.

[84] Hawick KA, Scogings CJ, James HA. Defensive Spiral Emergence in a Predator-Prey Model. *Complexity International* 2008; **12**:1–10.

[85] Hawick KA. Erlang and Distributed Meta-Computing Middleware. *Technical Report*, Computer Science, Massey University 2009.

[86] Cheng AHD, Cheng DT. Heritage and Early History of the Boundary Element Method. *Engineering Analysis with Boundary Elements* 2005; **29**(3):268–302.

[87] Fehlberg E. Low-Order Classical Runge-Kutta Fomulas with Stepsize Control and their Application to some Heat Tranfer Problems. *Technical Report TR R-315*, NASA 1969.

[88] Verner JH. Explicit Runge-Kutta Methods with Estimates of the Local Truncation Error. *SIAM Journal on Numerical Analysis* 1978; **15**(4):pp. 772–790.

[89] Dormand J, Prince P. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics* 1980; **6**(1):19–26.

[90] Prince P, Dormand J. High order embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics* 1981; **7**(1):67–75.

[91] Euler L. *Institutiones Calculi Integralis*, vol. 1. Carnegie Mellon University, 1768.

[92] Heun K. Neue Methoden zur approximativen Integration der Differential-gleichungen einer unabhängigen Veränderlichen. *Zeitschrift für Mathematick und Physik* 1900; **45**:23–38.

[93] Moore GE. Cramming More Components onto Integrated Circuits. *Electronics Magazine* 1965; **April**:4.

[94] Goth G. Entering a Parallel Universe. *Communications of the ACM* September 2009; **52**(9):15–17.

[95] Ghuloum A. Face the Inevitable, Embrace Parallelism. *Communications of the ACM* September 2009; **52**(9):36–38.

[96] Herlihy M, Shavit N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[97] Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, *et al.*. A View of the Parallel Computing Landscape. *Communications of the ACM* October 2009; **52**:56–67.

[98] Handy J. *The Cache Memory Book*. Academic Press Professional, Inc.: San Diego, CA, USA, 1993.

[99] Intel Coporation. *From a Few Cores to Many: A Tera-scale Computing Reserach Overview*.

[100] Intel Coporation. *Intel's Teraflops Research Chip*.

[101] Moore SK. Multicore is bad news for supercomputers. *IEEE Spectrum* 2008; **45**(11):11.

[102] Leist A, Playne DP, Hawick KA. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience* December 2009; **21**:2400–2437.

[103] NVIDIA® Corporation. *NVIDIA CUDA™ C Programming Guide Version 3.2* 2010. Last accessed December 2010.

[104] Baker M, Apon A, Buyya R, Jin H. *Encyclopedia of Computer Science and Technology*, chap. Cluster Computing and Applications. Marcel Dekker, 2001.

[105] Buyya R. *High Performance Cluster Computing: Architectures and Systems*, vol. 1. 1st edn., Prentice Hall PTR, 1999.

[106] Buyya R. *High Performance Cluster Computing: Programming and Applications*, vol. 2. 1st edn., Prentice Hall PTR: Upper Saddle River, NJ, USA, 1999.

[107] Duato J, Yalamanchili S, Lionel N. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2002.

[108] Andrews G. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999.

[109] TOP500org. TOP 500 Supercomputer Sites. `http://www.top500.org/`. Last accessed November 2010.

[110] IEEE. *IEEE Std. 1003.1c-1995 thread extensions* 1995.

[111] Intel. *Intel(R) Threading Building Blocks Reference Manual*. Intel, 1.16 edn. July 2009.

[112] Pacheco PS. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1996.

[113] Hempel R. The MPI Standard for Message Passing. *Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II: Networking and Tools*, HPCN Europe 1994, Springer-Verlag: London, UK, 1994; 247–252.

[114] Gropp W, Lusk E, Doss N, Sjkellum A. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*. Argonne National Laboratories, 1996.

[115] Burns G, Daoud R, Vaigl J. LAM: An Open Cluster Environment for MPI. *Proceedings of Supercomputing Symposium*, 1994; 379–386.

[116] Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, *et al.*. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004; 97–104.

[117] ATI. *Technical Overview ATI Stream Computing* 2009. URL `http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf`.

[118] Nguyen H. *GPU Gems 3*. First edn., Addison-Wesley Professional, 2007.

[119] Flanders PM. Effective use of SIMD processor arrays. *Parallel Digital Processors*, IEE: Portugal, 1988; 143–147.

[120] Hillis WD. *The Connection Machine*. MIT Press, 1985.

[121] NVIDIA® Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*.

[122] NVIDIA® Corporation. *Tuning CUDA Applications for Fermi*.

[123] NVIDIA® Corporation. *NVIDIA CUDA™ C Programming Guide Version 4.0* 2011. Last accessed May 2011.

[124] Dell. PowerEdge C410x Specification. online May 2011.

[125] Playne DP, Hawick KA. Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs. *Proceedings of International Symposium on Cluster, Cloud and Grid Computing*, Melbourne, Australia, 2010.

[126] Playne DP, Hawick KA. Comparison of GPU Architectures for Asynchronous Communication with Finite-Differencing Applications. *Accepted into: Concurrency and Computation: Practice and Experience* 2011; .

[127] Playne DP, Hawick KA. Asynchronous Communication for Finite-Difference Simulations on GPU Clusters using CUDA and MPI. *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2011.

[128] Xu T, Pototschnig T, Kuhnlenz K, Buss M. A High-Speed Multi-GPU Implementation of Bottom-Up Attention using CUDA. *2009 IEEE International Conference on Robotics and Automation*, Kobe International Conference Centre, Kobe, Japan, 2009; 41–47.

[129] Rizk G, Lavenier D. GPU Accelerated RNA Folding Algorithm. *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, Springer-Verlag: Berlin, Heidelberg, 2009; 1004–1013.

[130] Manavski SA, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 2007; **9**:1–9.

[131] Hartley TD, Catalyurek U, Ruiz A, Igual F, Mayo R, Ujaldon M. Biomedical image analysis on a cooperative cluster of GPUs and multicores. *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM: New York, NY, USA, 2008; 15–25.

[132] Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU Cluster for High Performance Computing. *SC Conference* 2004; **0**:47.

[133] Garland M, Grand SL, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V. Parallel Computing Experiences with CUDA. *IEEE Micro* 2008; **28**(4):13–27.

[134] Phillips JC, Stone JE, Schulten K. Adapting a message-driven parallel application to GPU-accelerated clusters. *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press: Piscataway, NJ, USA, 2008; 1–9.

[135] White JB, Dongarra JJ. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. *Proceedings of the 25th International Parallel & Distributed Processing Symposium*, 2011.

[136] Ray A, Kondayya G, Menon SVG. Developing a Finite-Difference Time-Domain Parallel Code for Nuclear Electromagnetic Field Simulation. *IEEE Transactions on Antennas and Propagation* 2006; **54**:1192–1199.

[137] Galbreath N, Gropp W, Gunter D, Leaft D, Levine D. Parallel Solution of the Three-Dimensional Time-Dependent Ginzburg-Landau Equation. *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[138] Xu M, Thulasiraman P. Parallel Algorithm Design and Performance Evaluation of FDTD on 3 Different Architectures: Cluster, Homogeneous Multicore and Cell/B.E. *High Performance Computing and Communications, 10th IEEE International Conference on* 2008; **0**:174–181.

[139] Jr JFS. Accelerating the Finite Difference Time Domain (FDTD) Method with CUDA. *Proceedings of Applied Computational Electromagnetics 2011*, 2011.

[140] Fox GC, Otto SW. Algorithms for concurrent processors. *Physics Today* 1984; **37**(5):50–59.

[141] Reed DA, Adams LM, Patrick ML. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers C* July 1987; **36**(7):845–858.

[142] Playne DP, Hawick KA. Visualising Vector Field Model Simulations. *Proceedings International Conference on Modeling, Simulation and Visualization Methods (MSV'09) Las Vegas, USA.*, 2009.

[143] ISO-IEC. *Working Draft, Standard for Programming Language C++* February 2011.

[144] ANSI-ISO-IEC. *Programming Languages—C++, ISO/IEC 14882:2003(E) International Standard*. Ansi standards for information technology edn. 2003.

[145] NVIDIA® Corporation. *NVIDIA CUDA Best Practices Guide 2.3*.

[146] Hawick KA, Leist A, Playne DP. Parallel Graph Component Labelling with GPUs and CUDA. *Parallel Computing* 2010; **36**:655–678.

[147] Parr T. *The Definitive ANTLR Reference - Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[148] Aho AV, Ullman JD. *Principles of Compiler Design*. Addison-Wesley, 1977.

[149] Levine JR, Mason T, Brown D. *LEX and YACC*. 2nd edn., O'Reilly, 1992.

[150] Paxson V, Estes W, Millaway J. *The Flex Manual - Lexical Analysis with Flex*. Version 2.5.35 edn. September 2007.

[151] Levine J. *Flex and Bison: Text Processing Tools*. O'Reilly Media, 2009.

[152] Hawick KA, Playne DP. Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA. *Concurrency and Computation: Practice and Experience* July 2011; **23**(10):1027–1050.

[153] Hawick KA, Playne DP. Automated and Parallel Code Generation for Finite-Differencing Stencils with Arbitrary Data Types. *Proceedings International Conference Computational Science, Workshop on Automated Program Generation for Computational Science*, Amsterdam, The Netherlands, 2010.

[154] Ising E. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift fuer Physik* 1925; **31**:253258.

[155] Wolf-Gladrow DA. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Springer, 2000.

[156] Merson RH. An Operational Method for the Study of Integration Processes. *Proc. Symp. Data Processing*, 1957; 110–125.

[157] Fehlberg E. Classical Fifth-, Sixth-, Seventh-, and Eighth-Order Runge-Kutta Formulas with Stepsize Control. *Technical Report TR R-287*, NASA 1968.