Data-parallel Structural Optimisation in Agent-based Modelling

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.

Alwyn Visser Husselmann
May 2014

Abstract

AGENT-BASED MODELLING (ABM) IS PARTICULARLY SUITABLE for aiding analysis and producing insight in a range of domains where systems have constituent entities which are autonomous, interactive and situated. Decentralised control and irregular communication patterns among these make such models difficult to simulate and even more so to understand. However, the value in this methodology lies in its ability to formulate systems naturally, not only generating the desired macroscopic phenomena, but doing so in an elegant manner. With these advantages, ABM has been enjoying widespread and sustained increasing use.

It is then reasonable to seek advances in the field of ABM which would improve productivity, comparability, and ease of implementation. Much work has been done towards these, notably in terms of design methodology, reporting, languages and optimisation. Three issues which remain despite these efforts concern the efficient construction, performance and calibration of agent-based models.

Constructing a model involves selecting agents, their attributes, behaviours, interaction rules, and environment, but it also demands a certain level of programming ability. This learning curve stymies research effort from disciplines unrelated to computer science. It is also not clear that one methodology and software package is suitable for all circumstances. Domain-specific languages (DSLs) make development much simpler for their application area.

Agent-based model simulation sometimes suffer tremendously from performance issues. Models of situations such as algal cultivation, international markets and pedestrians in dense urban areas invariably suffer from poor scaling. This puts large system sizes and temporally distant states out of reach. The advent of scientific programming on graphical processing units (GPUs) now provides inexpensive high performance, giving hope in this area.

It is also important to calibrate such models. More interestingly, the problem of calibrating model *structure* is given particular emphasis. This ambitious task is difficult for a number of reasons, and is investigated with considerable thought in this work.

In summary, the research shows that appropriate use of *data-parallelism* by *multi-stage programming* in a simple *domain-specific language* affords *high performance*, *extensibility* and *ease of use* which is capable of effective automatic model structure optimisation.

Key words: Agent-based Modelling, Domain-specific Languages, GPUs, Optimisation.

Acknowledgements

To the memory of David Frederik Husselmann Sr., 1929-2003.

The invaluable guidance of several people have made this project possible. Firstly, I would like to express my gratitude to my supervisors, Prof. Ken Hawick and A/Prof. Chris Scogings, without whom this dissertation would certainly not have been possible.

I am also greatly indebted to the members of the Complex Systems and Simulations Group at Massey University, especially Dr. Daniel Playne and Dr. Arno Leist for continuing advice and valuable suggestions for improvements to the manuscript.

I have been very privileged to have been able, in November of 2013, to marry my now-wife, Leah. To her I express my appreciation for her neverending support, especially during times of profound difficulty and late work nights.

My family has been very supportive, and I would like to thank my parents particularly, Erika and David, for a place to live and study. Without their support, I certainly could not have studied full time, and would not be in a position to progress my career for several years. My brothers, sisters-in-law and grandmother were also highly supportive during this time.

Finally, I would like to acknowledge the financial support I received from the Massey University Vice-Chancellor's Doctoral Scholarship.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Part I

Introduction

CHAPTER 1 OVERVIEW

The core theme across the content of this dissertation is the concept of Agent-based Modelling (ABM). Its widespread popularity and potency in complex systems analysis motivates much of this work. This chapter first presents the thesis structure and then introduces important concepts used throughout including ABM, Evolutionary Algorithms (EAs), and data-parallel computing with Graphical Processing Units (GPUs), Domain-Specific Languages (DSLs) and finally model calibration and optimisation.

The main objective is to make use of a staged programming methodology to leverage the computing power of GPUs in conjunction with a specialised DSL in order to accomplish pragmatic model structure optimisation for agent-based models.

1.1 Structure and Contributions

THIS DISSERTATION IS ORGANISED into four parts. Part I introduces the various concepts used throughout this work including EAs, Agent-based Modelling and Simulation (ABMS), data-parallel computing, and optimisation. Part II deals with continuous and combinatorial optimisation in the context of ABM and parallelism, and Part III introduces a DSL with built-in optimisation mechanisms using the techniques developed in previous parts. A photobioreactor model is also proposed, and used to demonstrate this new DSL. Finally, Part IV concludes this dissertation with discussions of findings, limitations, applications, opportunities for future work, as well as final conclusions.

The major original contributions in this dissertation are the following[1]:

1. A natural multi-stage domain-specific language for parallel agent-based modelling with a built-in model structure optimiser [126, 114] (see chapters 5 and 6).
2. A GPU-parallel modified implementation of the Firefly Algorithm by X.-S. Yang [299] for continuous global optimisation [118] (see chapter 3).
3. Use of Gene Scanning in Genetic Programming to propose a GPU-parallel Firefly Algorithm to operate in *program space* [128] (see chapter 4, particularly section 4.7).
4. An extended shallow-depth program-space visualisation algorithm, as well as a simple method for visualising large program trees [127, 128] (see chapter 4, particularly section 4.8).

[1]Some of these have been published by the author, or submitted, accepted and/or in press. Please see appendix A for a full list. Where these have been extended in the text, this has been noted.

5. An agent-based model of an algal photobioreactor based on the Photosynthetic Factory model [95, 113, 125] (see Chapter 7).

6. An investigation of the Karva language in conjunction with the Geometric Particle Swarm Optimiser [121] (see Chapter 4, Section 4.6.1).

Each chapter of this dissertation is self-contained, however it is suggested that Parts II and III be read sequentially. ($\mathcal{I}$, p. 3) Some suggested reading is provided by cross-referencing to relevant introductory material, which are provided by notes of the form ($\mathcal{I}$, p. 3) in margins. In this case, hypothetically referring to page 3 (an example is given in the margin here).

The rest of this chapter introduces core concepts and gives a general overview with respect to the major contributions listed above. First, the practice of ABM is presented, followed by EAs, data-parallel computing, DSLs and model calibration.

In the context of ABM (Section 1.2), EAs are discussed for their ability to calibrate parameters of a model in Section 1.3. However, their use is compute-bound and typically exhibit poor scaling characteristics. For this reason, data-parallel computing sees much use in this work, and is discussed in Section 1.4. Though the necessary apparatus is illuminated, the use of parallelism in the realm of agent-based model calibration is programmatically complex and difficult to accomplish. DSLs have in the past been useful to allow special syntax to express intention in a program using domain-specific knowledge, which is used in this dissertation to improve accessibility of the structural calibration mechanism for the end-user. DSLs are introduced in Section 1.5.

1.2 Agent-based Modelling

ABM is a modelling tool where components of a system are implemented as autonomous interacting nodes, or "agents" [24, 174]. Already, light has been cast on many complex systems which have been difficult to analyse in the past. Disciplines such as ecology [88, 87], microbiology [73, 65] and social science [59] benefit tremendously from the insight generated by this tool. Apparent self-organisation of birds and fish into flocks and schools, the foraging behaviour of ants and many other phenomena with no central source of instruction can now be explained by this method, and applications are increasing in number. The increasing popularity of the technique certainly warrants more investigation into its improvement.

There are various similarities in concept and implementation shared with other fields such as Population-based optimisation such as the Particle Swarm Optimiser [142], as well as Multi-agent Systems (MAS) [76]. Specifically, the presence of component-component interactions with a lack of central control.

The overall life cycle of the ABM process is shown in Figure 1.1. Firstly the researcher would begin with a hypothesis about a particular phenomenon, such as the spatial concentration of crime [23]. The modelling phase in this diagram follows the process detailed by Macal and North [174]. Implementing the model would simply follow the standard software development life cycle, which in the case of object-oriented programming would involve class diagrams and related documentation. Following the implementation phase, the model should be calibrated to ensure that it aligns as expected with the actual phenomena being studied. This model verification process is important to ensure the purpose of a model is consistent with its design [200].

Several aspects of ABM also make it particularly relevant in the realms of optimisation, parallel computing and domain-specific languages. Firstly, agent-based models are composed of "agents". Depending on the quantity of agents, their interactions can be parallelised to improve performance greatly [1] (discussed in Chapter 2, and

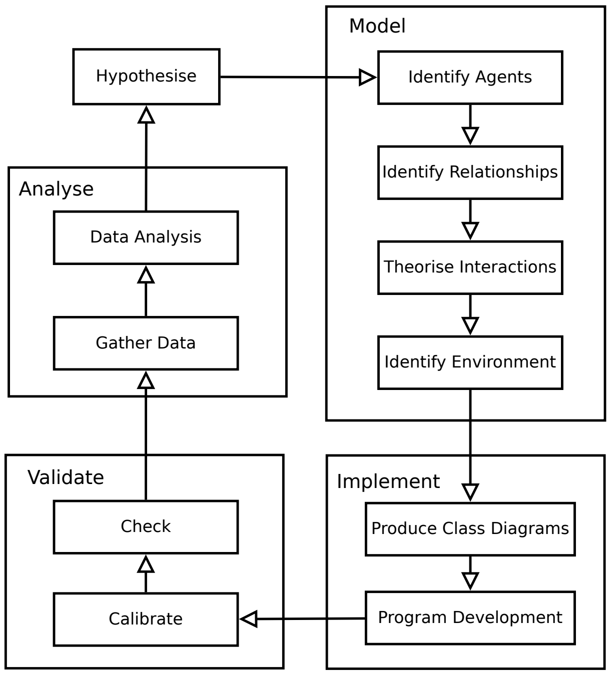


FIGURE 1.1: The high level Agent-based Modelling process. Starting from a hypothesis, a researcher would develop a theoretical model (shown according to Macal and North [174]), then proceed to implement it. Finally, the researcher would validate the model against the actual phenomena of interest, and then analyse the model quantitatively and qualitatively before the process begins again with a new hypothesis.

introduced in Section 1.4). Secondly, agent-based models often require a laborious parameter calibration effort, which can, under certain circumstances, be done automatically by re-interpreting the effort as an optimisation problem [28]. Finally, multi-disciplinary interest in ABM has demanded an easier implementation process, which has been shown by software packages such as NetLogo [279] to be something that can be mitigated by domain-specific languages. In the pursuit of this dissertation, some of these links were also explored and exploited to assist in the final objective.

With regard to the difficult and laborious calibration efforts that sometimes plague the use of ABM, there fortunately exist several automatic algorithms which can assist in the process. These are not trivial to construct, however, and may also suffer from performance problems themselves. A particular focus is given to evolutionary algorithms for their natural formulation and close relation to concepts within ABM. These are introduced in the next section.

1.3 Evolutionary Algorithms

Scientists in the field of numerical optimisation have looked towards many biological sources for inspiration including Fireflies [298] and Ants [48]. One technique which showed great promise very early on in 1975 [107] was the Genetic Algorithm (GA). EAs have become somewhat of a generalisation of the traditional GA, where the original process is modified, expanded, or replaced entirely. An excellent example of a sophisticated EA is Ferreira's Gene Expression Programming (GEP) [64].

Figure 1.2 is a flow chart showing the general process which EAs tend to follow in order to maximise (or minimise) an objective function. EAs maintain a population of possible solutions which are evolved over time using

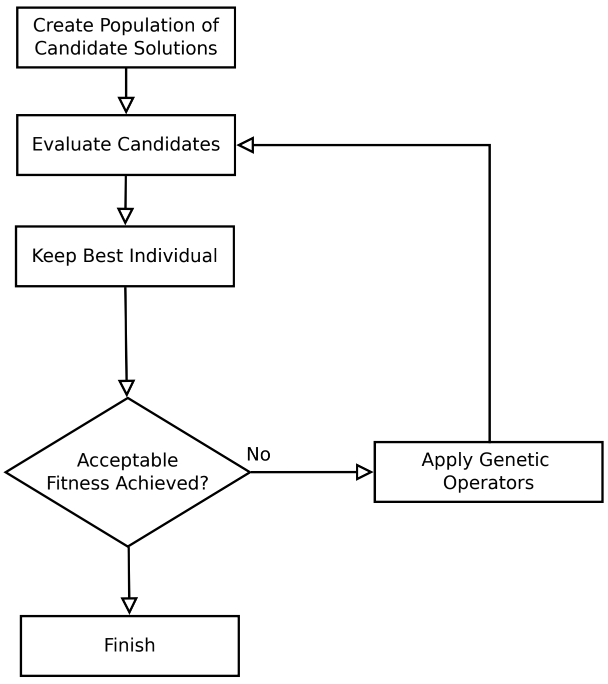


FIGURE 1.2: The typical process followed by an Evolutionary Algorithm. A population of candidate solutions is maintained, and evolved over time until an individual in the population reaches a certain fitness.

genetic operators such as mutation, crossover and selection. The process is analogous to that of nature, although, greatly simplified.

EAs have been shown to be effective for combinatorial [149, 187] and continuous optimisation problems [81], and consist of Evolutionary Strategies (ES), Evolutionary Programming (EP) and Genetic Algorithms (GAs) [11], the purpose of which is to optimise some real-valued objective function.

Optimisers based on EAs are well suited to the problem of model parameter optimisation, as well as model *structure* optimisation due to relaxed definitions of the meanings of concepts such as distance. However, these algorithms still involve a number of individuals in a population which must all be evaluated. Sometimes these population sizes can be very large or extremely computationally expensive to evaluate, or both. This presents an opportunity to exploit another inherent parallelism for improving quality of solutions and speed.

A particularly useful and recent development in parallel computing is that of general-purpose graphics processing units (GPGPU). These devices have extremely high theoretical throughput, which can be fully exploited only by a carefully designed algorithm. Fortunately, this has become easier with the advent of NVIDIA's Compute Unified Device Architecture (CUDA). This is discussed in the next section.

1.4 Data-parallel Scientific Computing

GPUs have recently become the focus of attention for many researchers requiring more computing power than what a typical desktop machine offers [162]. The reason for this is mostly due to the remarkably low cost of these devices, considering the performance a carefully designed program can achieve on them. The history of the general-purpose use of GPUs did not begin with full vendor software support, however. In the beginning, they could only be utilised by writing specialised assembly instructions by hand, which was followed by higher level shader languages [161]. NVIDIA later released CUDA [202], along with other vendors releasing similar software. The ease with which these devices can now be utilised makes it clear why scientific computing on GPUs is increasing rapidly [161].

Continuous global optimisation algorithms benefitted quickly from this technology, as shown by several authors [156, 157, 304, 128], including the fascinating work of Cupertino and colleagues [41] on evolving GPU PTX instructions directly. The ABM community also gained FLAME GPU [244], with several authors proposing GPU-variants of Agent-based Models [1, 287, 125, 51].

GPUs are by no means the only parallelisation strategy one may approach. Multi-core processors should certainly not be ignored as well as data-parallel extensions such as SSE and AVX [68], as shown by Chitty [36]; and of course, cluster computers are still applicable [1]. For commodity pricing, GPUs certainly bring a large amount of processing power to the workstation of today's researchers.

Though CUDA has made the use of GPUs much easier, it is not always obvious how one might utilise a GPU in arbitrary simulations. There has been research towards automatic parallelisation in the past, and has typically made use of domain-specific languages (DSLs) to expose this functionality [221, 244, 247]. DSLs are introduced in the next section.

1.5 Domain-Specific Languages

A DSL is a programming language (interpreted or compiled) which serves within a certain, narrow target domain [269]. Efficiency in the writing of code can be improved by using such a language [99]. The advantages of using custom languages include concise code with a clear intention which is easier to understand and easier to write. In addition, experts in the domain may not always be experts in programming. In these cases, a language which applies naturally to the target domain will greatly assist. DSLs have been applied to ABM in the past, with a prominent example being NetLogo [279]. Other ABM-related examples include the work of Franchi [70] and Hahn [91]. DSLs are not new, but have received greater attention in recent years [269, 182].

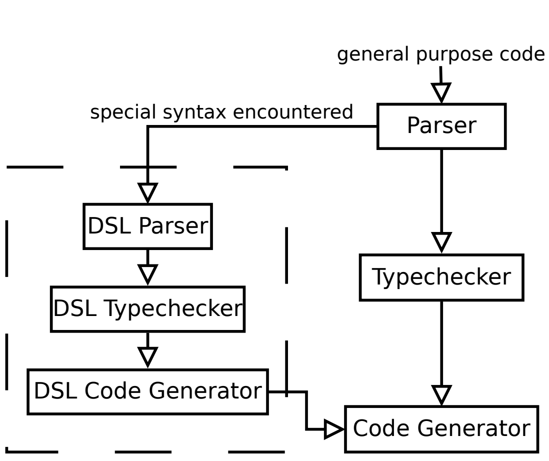
Taha [273] notes the characteristics of DSLs as having:

1. A clearly defined target domain
2. Notation with a clear intent
3. Clear formal *and* informal semantics

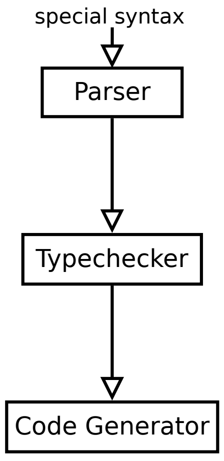
Figure 1.3 shows example compiler architectures. These are by no means the only architectures, but are most relevant to this work. Both Figures 1.3(a) and 1.3(b) show statically typed languages. What separates internal and external DSLs is that an internal DSL is supported, created and managed by a general purpose language, such as C++. Whereas in an external DSL, the language is completely standalone, and much more effort is normally involved.

An example of a DSL for controlling a state machine might be that shown in Listing 1.1. Though this is only a hypothetical example, the advantages of such a language would be very beneficial, should the expert using the language be less familiar with programming. It affords a clearly recognisable intent which makes it much more conducive to cooperation among experts than obscure code written in a powerful general purpose language.

DSLs are investigated in this dissertation for their ability to simplify a more complex algorithm, such as a model parameter or structure optimiser. By hiding complexity from the user, a more usable library can be developed. This idea is not without its disadvantages however. This is discussed in detail in Chapter 5.



(a) Internal DSL architecture.



(b) External DSL architecture.

FIGURE 1.3: The architectures of typical compiled internal and external DSLs. External DSLs essentially follow the typical compiler architecture, except that it is newly made for a specific purpose. Internal DSLs benefit from a (usually) pre-defined general purpose language, which is simply extended by an inner DSL.

```
I = compute_illumination_coef()
:activated_state
with probability b*I
  goto inhibited_state
:inhibited_state
with probability d
  goto resting_state
:resting_state
with probability a*I
  goto activated_state
```

LISTING 1.1: An example hypothetical DSL for controlling a state machine operating on a Markov Decision Process.

Given a domain-specific language (this section) to ease parallelisation (Section 1.4) of optimisation by evolutionary algorithms (Section 1.3), agent-based model calibration is made accessible, sufficiently fast, and effective. One final aspect which was investigated was *structural optimisation*, made possible by the same set of concepts, aimed towards combinatorial optimisation. The task of continuous and combinatorial optimisation in model calibration is discused in the next section.

1.6 Model Calibration and Optimisation

Optimising the parameters of an agent-based model has been done by Genetic Algorithms [28, 250] and other algorithms such as "Adaptive Dichotomic Optimisation" [29]. As long as a model can be measured in terms of how well it satisfies an objective, an optimiser can be applied to calibrate parameters, as well as behaviour. Junges and Klügl have investigated the optimisation of behaviour [137, 136, 134] using a variety of algorithms including some machine learning techniques and algorithms from decision tree induction, as well as Genetic Programming [149], with varying success.

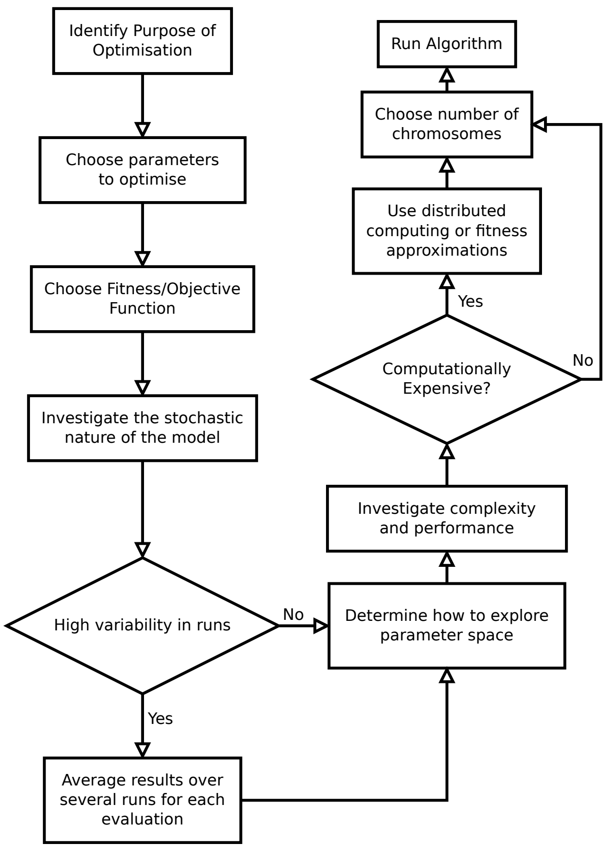

Identify Purpose of Optimisation
Choose parameters to optimise
Choose Fitness/Objective Function
Investigate the stochastic nature of the model
High variability in runs
No
Yes
Average results over several runs for each evaluation
Determine how to explore parameter space
Investigate complexity and performance
Computationally Expensive?
Yes
No
Use distributed computing or fitness approximations
Choose number of chromosomes
Run Algorithm

FIGURE 1.4: The process a reseacher would follow to calibrate a model using a continuous global optimiser, based on the process outlined by Calvez and Hutzler [28] for Genetic Algorithms. Choosing the objective function is a particularly difficult task, and then one must further decide whether to average the results to eliminate stochastic influence. Finally, the parameter set of the optimiser must be satisfied, and computationally expensive simulations need to be dealt with.

Figure 1.4 outlines model calibration by continuous global optimisation. Several aspects of this are worth noting. Firstly, choosing an objective function is not trivial, and has a very strong influence on the final result. Secondly, the process has the potential to become very computationally expensive, very quickly. A model simulation might already be expensive to compute. For instance, a model with $n_{\text{agents}} = 1000$ interactive agents. How many time steps n_t to compute for one run is also a question which only the researcher can answer. Suppose then that there is some stochastic influence, which forces the result of every run to be averaged n_{av} times. Further, for a population-based optimiser there must be a set of n_p candidate model simulations.

Viewing an Agent-based Model and its parameters as an optimisation problem brings with it some computational issues [28]. Firstly, an Agent-based Model is typically expensive to compute. Suppose a model has n agents, and every frame, each agent must communicate with every other agent. This is at least $n(n-1)/2$ interactions, and in simple implementations, simply $n(n-1)$. In other words, such a simulation has complexity $\mathcal{O}(n^2)$.

Depending on the model, not all of these interactions might be necessary [119], and can be eliminated, as in the case when two agents only interact when they are within communication distance. As shown by the author, it is not trivial to eliminate this complexity, since each agent must still communicate in order to compute their relative distance [119]. It is possible to use spatial partitioning techniques to reduce the complexity.

In order to accomplish the objective of automatic model structure optimisation, a suitable optimiser is necessary. It is interesting to note that there are similarities in some population-based optimisers to ABM. For instance, the Particle Swarm Optimiser (PSO) [142] maintains a population of real-valued candidate solution vectors which interact in order to move closer to the global optimum. The Firefly Algorithm [299] is similar to the PSO, except that its candidates observe others differently in the spirit of light intensity decay through a medium. Similar complexity issues arise also, from interacting agents in ABM. Effectively, this allows us to view population-based optimisers with interacting candidates as agent-based models, and similar data structures and algorithms can be applied to improve their performance. Moreover, ideas and techniques from population-based optimisation can be superimposed on ABM. This opens the door to inter-discipline synergy once more.

Using parallel computing is of particular interest in this work and the realm of ABM in general. Some models (as shown above) have the potential to become very computationally expensive, and their optimisation even more so. The use of graphics hardware to improve performance in this area has shown to be very effective, and is discussed in detail in Chapter 2. It is further discussed in Chapter 6.

1.7 Summary

The core concepts used throughout this dissertation were introduced and discussed. These included agent-based modelling, evolutionary algorithms, data-parallel scientific computing using graphics processing units, domain-specific languages and finally model calibration which uses all the core concepts to its advantage.

There are some interdependencies among these also discussed in this dissertation, such as larger system sizes in both agent-based models and population-based evolutionary algorithms being both facilitated using graphics processing units (see Chapters 2, 3 and 4). Domain-specific languages are also useful not only for easing parallelisation, but also for ABM in general (see Chapter 7), as a specific target domain. In addition, calibration is a problem not only faced by ABM, but also by optimisers themselves. The process of calibrating a population-based evolutionary optimiser is essentially equivalent to calibrating an agent-based model, and this is investigated in detail in Section 3.5.

"People think mathematics is complicated—mathematics is the simple bit. It's the stuff we *can* understand. It's *cats* that are complicated. What is it in the molecules that makes one cat behave different from another cat? How do you define a cat?"

John H. Conway

CHAPTER 2 PARALLEL AGENT-BASED MODELLING AND SIMULATION

Agent-based Modelling (ABM) is an exciting field of work. We are currently entering a phase where scientific inquiry in certain complex systems is being bolstered significantly by generative modelling methodologies, where macro-level phenomena are created "bottom-up". The advantages here are certainly not without drawbacks, however. One area which is of particular interest in this work is that of system size scaling. Increasing the scale of an interactive agent-based simulation invariably leads to poor performance, and at times, to the point where further investigation is stymied, or simply impossible. We are fortunate to have experienced significant advances in data-parallel scientific computing in recent years. Unfortunately, these are not trivial to harness. This chapter offers a detailed introduction to ABM in Section 2.1 followed by an introduction to the parallelisation of these on Graphical Processing Units (GPUs) in Section 2.2.

Code examples are given in a language code-named "MOL", a new domain-specific language for Agent-based Models which is described in detail in Chapters 5 and 6.

The contents of this chapter extend work previously published by the author, in the *IIMS Postgraduate Student Conference Proceedings*[1], *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems*[2] and also *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods.*[3]

2.1 Introduction to Agent-Based Modelling and Simulation

THE TECHNIQUE OF ABM HAS REACHED A MILESTONE in recent years as a well-developed research methodology in terms of practice [62, 47] and theory [185]. The work of Nobel laureate Thomas Schelling in 1971 [253] set in motion the initial concept of ABM in social science, as credited by many [174, 287, 175, 23, 59]. The concept of decentralised control was already under investigation, given that John Conway's revolutionary Game of Life was published in 1970 by Gardner [75], which popularised cellular automata and artificial life with similar concepts. The simplicity with which Conway's Game of Life created complex macroscopic

[1] A. V. Husselmann and K. A. Hawick. Spatial agent-based modelling and simulations - a review. Technical Report CSTN-153, Computer Science, Massey University, Albany, North Shore,102-904, Auckland, New Zealand, October 2011. In Proc. IIMS Postgraduate Student Conference, October 2011

[2] A. V. Husselmann and K. A. Hawick. Simulating species interactions and complex emergence in multiple flocks of Boids with GPUs. In T. Gonzalez, editor, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 100–107, Dallas, USA, 14-16 Dec 2011. IASTED

[3] A. V. Husselmann and K. A. Hawick. Spatial data structures, sorting and GPU parallelism for situated-agent simulation and visualisation. In *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods (MSV'12)*, pages 14–20, Las Vegas, USA, 16-19 July 2012. CSREA

patterns interested many. Like Schelling, Conway did not use a computer, rather they used chessboards, graph paper and coins.

The practice of ABM stems from earlier works in complex adaptive systems (CAS) [174], which are generally concerned with the investigation of (usually biological) decentralised systems where constituent parts interact in some manner. John Holland, inventor of the hugely popular Genetic Algorithm [107], was very influential in the CAS literature [108].

Conway and Schelling's works were followed by prominent works of authors such as Robert Axelrod [6]. During this time, an influx of scientific attention in this field surfaced. Agent-based Modelling became the name for a somewhat expanded investigation of complex adaptive systems and related phenomena in a wide range of disciplines [174]. Reynolds elegantly demonstrated precisely what agent-based models are capable of [240, 241] towards the latter years of the 1980s. His model showed realistic flocking behaviour, with no central source of control. At that time it became prudent to examine ABM more formally, which resulted in some success [236]. The lack of central control and successive interaction in these models makes mathematical analysis (and hence formalisation) difficult. There has, however, been research in fields which share similar reactive-agent complexity, such as population-based global optimisation [285]. For the time being, statistical analysis and experimentation are the best tools for understanding agent-based models [233]. Further research continued in earnest in the 1990s with the very prominent works of Axelrod and Epstein [7, 60], which cemented the use of ABM in the social sciences.

The uptake of ABM in social science [59] was more or less concurrent with that of ecology [89], albeit with different terminology. In ecology, ABM was effectively named "Individual-based Modelling" (IBM) which has in latter years been identified as equivalent to ABM [87, 233]. Similar concepts were in use, albeit with different terminology suited to the individual discipline. A great deal of research effort has from early years been invested in IBM [87]. Railsback and Grimm in their work of 2012 use the terms Agent-based Modelling and Individual-based Modelling interchangeably [233], in some ways signifying inter-discipline unification; but it is worth noting that in history, IBM has mostly been discussed in terms of ecology.

During the 1990s, while ABM was still spreading as a methodology [105], it started being perceived as complete opposition to traditional methods [215] such as partial differential equations. In general, the generative, "bottom-up" approach of ABM makes it naturally applicable to systems with interactive entities. It is perhaps this natural application (notably different from equation-based approaches) which made ABM be perceived as a "...'third way' of carrying out social science," [80]. A recent article in the *International Journal of Business and Economics Research* by Voicu and Galalae indicates in an opening statement that ABM and equation-based models are essentially mutually exclusive [287]. Bonabeau however, makes the case that a model with differential equations imposed upon its individual components *is also* an agent-based model [24]. All of these points considered, ABM should be considered as another complementary piece of apparatus, instead of a competing alternative. There are still situations where ABM is not applicable, especially when the purpose of a system *is* central control or a deliberate lack of communication between constituent entities.

The overall efficacy of ABM as a technique for analysing phenomena with decentralised control and interactive entities has attracted great interest from many research fields; and many models have been reported for simulating a wide variety of phenomena, including social interaction scenarios [174, 8], crowds [163, 140], vehicles [77, 277], robotics [261], predator-prey models [104, 257, 133], biological phenomena [73], and models casting light on flocking behaviour [240, 241], as well as archaeology and anthropology [174]. ABM has even found use in cancer immunology [66, 67]. Even though such widespread success has arisen, some issues remain, which stifle further research built upon works of others.

Due to the multi-disciplinary nature of ABM, it has conflicting definitions of key concepts. This is damaging when trying to compare models in the same domain, or even trying to interpret results; not to mention that it stymies multi-disciplinary cooperation. The concept of an "agent" for example, has been a contested concept for some time [174]. Fortunately however, most authors tend to agree on key aspects [174]. Drogoul, Vanbergue and Meurisse disagree, however, insisting that this consensus regarding terminology is "at best, syntactic" [50]. During the time which ABM was not yet fully interdisciplinary, it seemed reasonable to define terms which suit the application domain at hand. For example in economics, Das asserted that agents have some level of bounded rationality [45]. Ferber [62] concedes that the term "agent" is used with little or no definition in some disciplines, and proceeds to give a list of 9 attributes which define an agent. In the future, it seems more probable that a hierarchy of theory will present itself, with a general theoretical knowledge base, with more specific specialisations and formalisms in specific domains. Theoretical apparatus such as Belief-Desire-Intention agents [237] and the X-Machine formal agent description [37] gives hope in this area.

There has been much confusion with the definition of the term "agent", as exemplified by a remark of Carl Hewitt[4] who pointed out the equivalence between the question "What is intelligence?" in the AI community and "What is an agent?" in the ABM community. It seems that the best (albeit not universally optimal) definition can be constructed from the most widely used attributes and qualities of relevant models. The recent work of Macal and North identifies five characteristics that seem to be the most common [174] in recent literature[5]:

1. Identity - Agents must be identifiable, discrete individuals
2. Situation - Agents are situated in some fashion
3. Goal-oriented - Agents have goals to achieve
4. Autonomy - Agents are autonomous and may operate independently
5. Learning - Agents could potentially learn and adapt

Emergence is another contested term in interaction-based simulations[6]. Reynolds' flocking simulation [241] demonstrated the ability of three simple local interaction-based rules which generate perhaps inexplicable complex patterns. Conway's Game of Life [75] also caused macroscopic complexity to emerge, which was not directly specified. These features were termed "emergence". Epstein notes that the term was often mystified during times before complex systems were dealt with [59], and notes an early use of the term in the example of bond formation between Oxygen and Hydrogen atoms, which was often assumed to be fundamentally unexplainable [59].

At present, emergence is certainly still a contested term. However, from a pratical point of view, it is typically used to refer to macro-level patterns of behaviour which were not necessarily directly engineered, or noticeable in designed local interaction specifications [60]. Figure 2.1 shows an example of what such simple local interaction rules can generate. Each agent has simple reactive rules depending on the spatial configuration of other agents in its immediate neighbourhood. The link between emergence and autonomy of interactive agents seem undeniable. It would be extremely difficult to generate dynamic macroscopic patterns such as these using a central controller. Macy and Willer in 2002 remarked that if Reynolds had instead pursued a central controller or "top-down" algorithm for flocking behaviour, "he might still be working on it" [175].

[4]At the 13th Int. Workshop on Distributed AI, cited in Wooldridge and Jennings' work, *Intelligent Agents: theory and practice* [295].

[5]This definition of agents is adhered to as much as possible in this work.

[6]As with other terms, the term "emergence" is also poorly defined, and perhaps altogether inappropriate [59].

There are other methodological issues which arise in the use of ABM in practice. For instance, there is a certain level of programming knowledge that is necessary for researchers to use ABM to their advantage [212]. Many software packages have emerged over the last decade to attempt to allow re-use of knowledge and reduce the steep learning curve associated with custom-developed software. These include platforms such as RePast [38], SWARM [186, 276], MASON [169], FLAME GPU [244], and NetLogo [279], among others. Each of these software packages have their own merits. Railsback et al. compared these in detail [234].

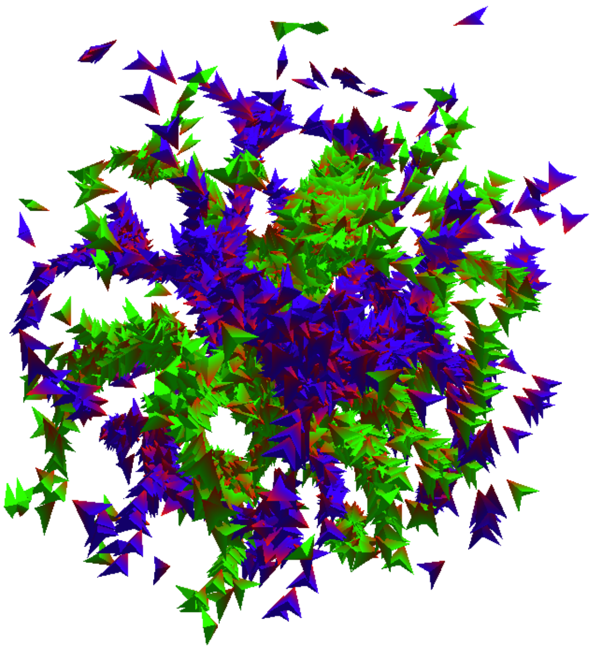


FIGURE 2.1: An example of emergence: macroscopic patterns which are seemingly unrelated to underlying interaction rules emerge. This image was generated using a modified Flocking simulation.

There is also the consideration of implementing agent-based models, and methods vary considerably. There have been models where agents are represented by lists of rules [258], finite state machines [125] and situated in continuous space [115] or discrete lattices [113]. From a practical standpoint, the details of implementation are certainly not to be ignored. Sometimes these specifics can have a considerable effect on the model itself, which was a concept which made many early researchers in the field of ABM concerned and uneasy [89]. Now, there are several methods of implementation, and each is more suited to certain domains than others. Proto [18] for instance, is specific to the field of spatial computing and sensor networks, but its paradigm can also be used for ABM [282]. For more general agent-based models, NetLogo [279] contains a special limited Domain-Specific Language (DSL) which is used to specify agent behaviour, whereas in RePast [38] agents are defined using Java, a general purpose language. The spatial configuration of agents is also important [174]. As for how agents are situated, Figure 2.2 shows a selection of configurations. A focus is given to discrete lattice spaces, where individual sites are able to contain an agent which is free to move to other sites and interact with other agents.

Implementation and frameworks aside, by considering agent-oriented systems already discussed in the literature, one can obtain an appreciable idea of the wide variety of models that have been created which satisfy (at least partially) the most common definitions of ABM concepts and certainly also gain a sense of precisely how flexible ABM is.

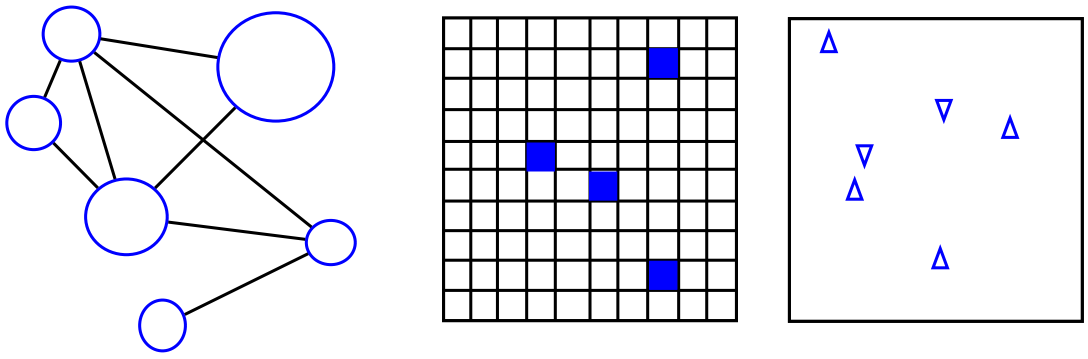


FIGURE 2.2: Spatial topologies of agent-based models: network, lattice and continuous-space. Models based on these different topologies vary considerably.

It is prudent to further the introduction on ABM with suitable examples. In the following sections, Conway's Game of Life is explained in detail, followed by Reynolds' "Boids" simulation, the Predator-Prey model and a short discussion on spatial computers. Conway's Game of Life is formulated on a "lattice", or checkerboard, whereas Boids is formulated in continuous space. The Predator-Prey model discussed here is formulated on a lattice, though it can also be formulated in continuous space. Spatial computer nodes communicate by local radios. All of these are excellent examples of how simple local interaction rules can lead to complex macroscopic behaviour through successive agent-agent interactions in different spatial configurations.

2.1.1 Conway's Game of Life

The pioneering work of Conway resulted in a fascinating "mathematical game" in 1970, published by Martin Gardner [75]. Conway's work was predominantly in pure mathematics. The "Game of Life" resulted from a past time of Conway which was what Gardner called "recreational mathematics"; the results of which Conway almost never published. Even in recent years, Conway's Game of Life still inspires research [103, 94].

The game is played on a large checkerboard with counters ("cells"), starting with a random or predetermined number of live, and empty cells. The rules are extremely simple and elegant, and were given by Gardner as:

1. "Survivals". A cell survives if it is neighboured by two or three other cells.
2. "Deaths" (overcrowding). A cell dies if it is neighboured by four or more cells.
3. "Deaths" (isolation). A cell dies if it is neighboured by one or zero cells.
4. "Births". If an empty cell has exactly three live neighbouring cells, it will become a live cell.

Figure 2.3 shows 14 time steps (or "generations") of the Game of Life starting with a simple hand-made pattern. The initial configuration becomes stable after 13 steps. It is certainly difficult to predict without simulation whether a pattern will become stable, and in what configuration. These are two of the objectives Conway had in mind while designing the rules of the game [75].

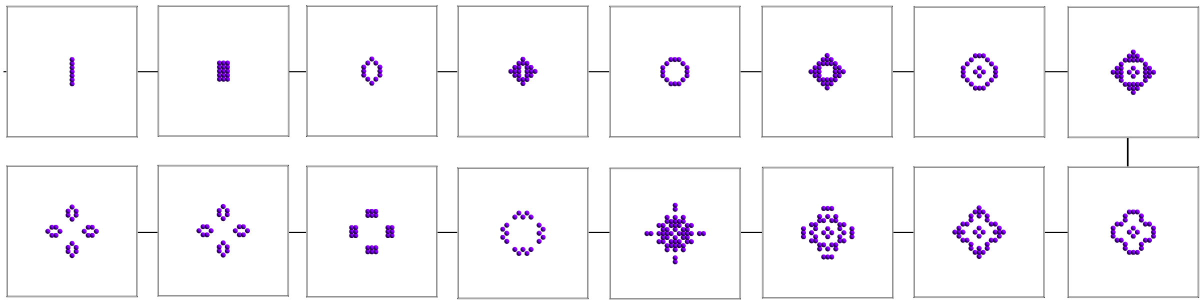


FIGURE 2.3: An example of Conway's Game of Life with an initial pattern that becomes stable after 13 steps.

```
defvar c = get_neighbour_count
 if  current_site  == 0 then
     if  c == 3 then
         comealive
     end
else
     if  c >= 4 then
         die
     elseif  c <= 1 then
             die
         end
     end
end
```

LISTING 2.1: A simple implementation of Conway's Game of Life, written in the MOL language (see Chapters 5 and 6).

The code shown in Listing 2.1 generated the images shown in Figure 2.3. This language is discussed in great detail in Chapter 5. The same code is executed for every site in a lattice of 32 by 32 cells, where a value of 0 indicates a dead cell, and 1 indicates a live cell. What makes this directly related to ABM is the fact that cells are autonomous and local interactions decide their fate. Although cellular automata is different in purpose and general formulation to the practice of ABM, it embodies some of the most basic apparatus which ABM relies on fundamentally: autonomy, interaction, spatial situation, and also emergence. For this reason, it is included here as part of introduction to ABM. Emergence in the Game of Life refers to the overall macroscopic patterns which are not predefined, but rather, they emerge from the predefined rules for each individual cell.

Conway's Game of Life takes place on a lattice, therefore, it is also relevant to discuss agent-based models which operate in continuous space. The work of Reynolds on flocking simulations is an ideal example of this, and is discussed in the next section.

2.1.2 Flocking Simulations

Agents in ABM are typically represented by a set of rules for their behaviour, or a decision tree, or a finite state machine. For Conway's Game of Life, rules are enforced simply by testing a list of rules and executing the first action which has a satisfied precondition. In the case of the "Boids" simulation, each agent is endowed with three simple deterministic vector-based rules [240]: cohesion, alignment and separation, which are each applied in turn without fail (except where no agents are in communication distance). The original Boids algorithm is summarised in Algorithm 1.

```
initialise $N$ boids in continuous space.
for $i \leftarrow 0$ to $i_{N-1}$ do
    gather agents into set $C$ that are within communication radius of agent $i_N$
    Rule 1: compute relative vector to centre of mass $V_c$ of boids in $C$
    Rule 2: compute average velocity vector $V_v$ of boids in $C$
    Rule 3: compute separation vector $V_s$ of boids in $C$ which are in the separation radius
    new $V \leftarrow V_0 + V_s + V_c + V_v$
end for
```

ALGORITHM 1: The original Boids algorithm by Reynolds [240].

Apart from these rules, agents have bounded knowledge of their surroundings and the environment. Each agent may only perceive of other agents within a certain radius, and a view restricted to a forward-facing spherical sector. Bounds on knowledge of an agent is important not only in simulations such as these, but also in economics [45].

In summary, the rules of the original Boids model are [240]:

1. “Separation”: Avoid crowding local agents.
2. “Alignment”: Attempt to match the average heading of local agents.
3. “Cohesion”: Tend towards the centre of mass of local agents.

By maintaining a velocity for each agent, one simply adds a vector contribution of each of these rules. The result of which, in 1987, was astonishing [240], and was described as “emergence” for emerging from the successive complex interactions within a group of autonomous agents. These rules do not obviously indicate the macro-level behaviour associated with them. Common across the rules above is that they all involve only local agents: agents within a certain distance (and heading as in the original). But when all agents interact with their local counterparts, they create a global pattern, which at first, seems to indicate a collective unity; as if there was only one agent. This is indeed noticeable in nature, when fish school together and sometimes assume the movements and shape of a much larger fish [76]. Attempts to take advantage of this led to the concept of *holonic* multi-agent systems, which Gerber, Siekmann and Vierke define as a multi-agent system where certain agents deliberately lose some of their autonomy in order to become part of a “super-agent” or “holon” [76]. Multi-agent systems are closely related to ABM, but more directly to distributed artificial intelligence [50, 62].

Figure 2.4 shows an example of a modified Boids simulation with multiple flocks, identified by their respective colours. An analysis of this model is available [115]. The difference between this model and the original Boids model is simply ignoring nearby agents which are of foreign flocks. Nevertheless, interesting behaviour ensues when these flocks interact, especially when they repel, rather than simply ignoring each other. This is yet another example of emergence, though the outcome from such a simulation is perhaps more easily predicted.

It is also necessary to consider precisely how computationally expensive such a model is to execute. An immediate application of this algorithm is crowd simulation [140, 241] in films. A number of agents such as 2000 in Boids would begin to stretch the limit of a single-threaded implementation. It is not realistic to ignore the

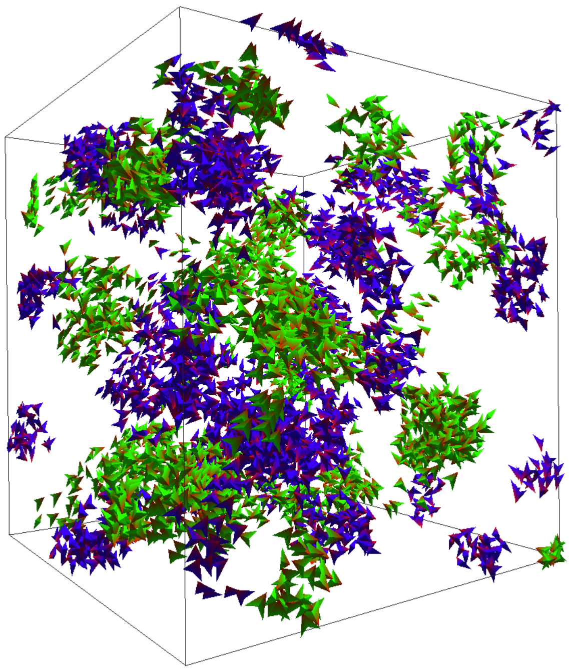


FIGURE 2.4: The Boids model with separate flocks which repel foreign agents and attract like agents.

possibility that one may require a crowd much larger, perhaps 10^6 or even more. Section 2.2 deals with this in detail.

Another example is the Predator-Prey model [104, 257]. This model is interesting for several reasons, including its application to ecology, as well as its ability to benchmark some machine learning algorithms. This model is discussed in the next section.

2.1.3 Predator-Prey Models

The Predator-Prey model is essentially a “food chain” simulation which stems from the independent works of Lotka and Volterra early in the 20th centuary [166, 288]. There are many varieties of these simulations with small differences in formulation, but in general they are helpful for ecological inquiry as well as benchmarking machine learning algorithms [171]. Important discoveries have also been made, such as the fascinating work of Jim and Giles, in which they discovered, quite simply, that “talking helps” [133]. By which the authors meant that when predators communicate using a simple language, they can help coordinate their efforts successfully.

In Predator-Prey models, two types of individuals exist, named predators and prey. Predators pursue and kill prey, and prey attempt to escape from the predators. Typically, this is formulated on a closed lattice or in continuous space, frequently with periodic boundaries. Predators breed depending on how successfully they kill the prey, and/or spatial proximity to other predators, or simply by a constant probability. The precise formulation of the model depends on its purpose. There have been studies on food webs [102, 232], multiple species [101, 100], spatial emergence [104] and altruism [257].

Figure 2.5 shows a plot of the proportion of prey and predators in the lattice. It is interesting to note that, with some random variation, the difference between the curves is essentially amplitude and phase, even though only one species can consume the other. The formulation of this simulation was very simplistic. The code for this model is shown in Listing 2.2. As with Conway’s Game of Life, this particular implementation is formulated on a square

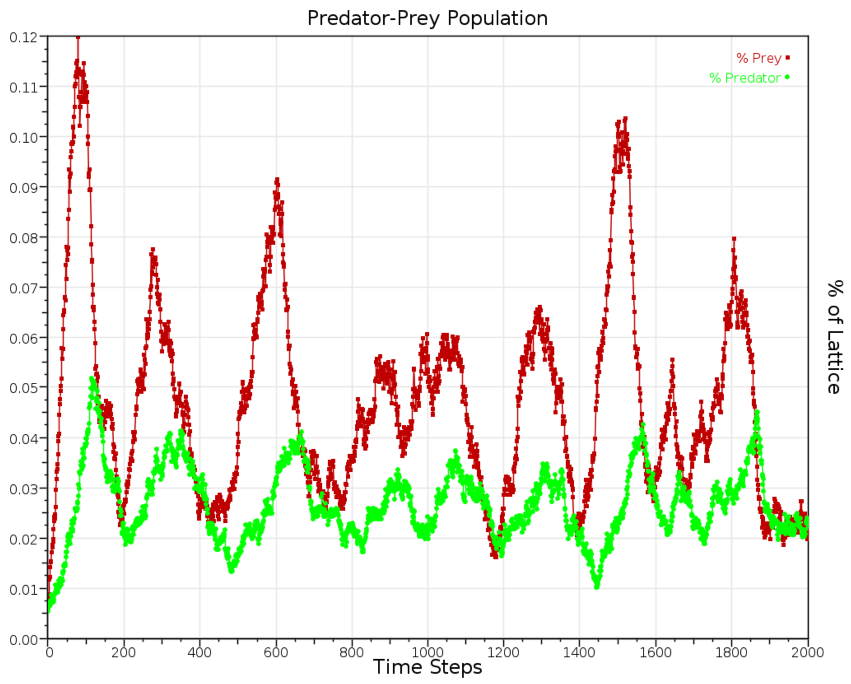


FIGURE 2.5: A plot of the predator and prey lattice coverage. Interesting in this plot is that the number of prey and predators essentially only differ by amplitude and phase.

lattice, but 64 by 64 cells. Each cell can contain either a predator, or a prey animal (these "artificial animals" are termed "Animats" in many Artificial Life publications [101, 257]). A summary of the rules, as implemented in Listing 2.2, is given in Tables 2.1 and 2.2. A rule is tested and executed if its precondition is satisfied, otherwise, the next rule is evaluated, until all rules have been tested, or one rule has been executed.

Rule	Precondition	Action
1	Surrounded by 4 other predators	Die
2	*random number* < 0.1	Breed
3	*true*	Move towards closest prey

TABLE 2.1: A set of predator rules for the simple Predator-Prey model.

The autonomy and local interaction of the predators and prey classify the model as agent-based. By the definitions of Macal and North [174], predators and prey have *identity*, being discrete individuals. They are also *situated* on a lattice, and are clearly *goal-oriented*. They behave independently of each other, on their individual spatial arrangement, therefore they have *autonomy*, and finally, they do have potential to *learn*, depending on how the model is formulated. For example, when using the Predator-Prey model for evaluating machine learning algorithms, predators can learn to communicate [133]. Simpler formulations such as the one described here, do not include learning.

Again, the purpose of a model determines how its analysis will proceed. In this case, the numbers of the different species (predators and prey) are collected. Quantitative analysis is important, but often one can obtain useful insight from qualitative sources. Visualisation is a very good source for obtaining feedback during the

Rule	Precondition	Action
1	Closest predator is adjacent and 10% chance	Die
2	A predator is close	Move away
3	Surrounded by 4 prey	Die
4	*random number* < 0.3	Breed
5	*random number* < 0.5	Move randomly

TABLE 2.2: A set of Prey rules for the simple Predator-Prey model.

```
mol
  defvar count = 1
  defvar pred = 1

  query neighbours6
    if neighbour == 1 then
      count = count + 1
    else
      if neighbour == 2 then
        pred = pred + 1
      end
    end
  done

  defvar predator = 2
  defvar prey = 1

  if me == predator then
    if pred >= 4 then
      die
    else
      if randomfloat < 0.1 then
        breed
      else
        defvar temp = get_closest_prey()
        move towards temp
      end
    end
  end

  if me == prey then
    defvar closestpredator
      = get_closest_predator ()
    if (distance to (closestpredator)) < 2 then
      if randomfloat < 0.1 then
        die
      end
    else

      defvar closestprey = get_closest_prey ()

      if (distance to (closestpredator)) < 3 then
        move awayfrom closestpredator
      else
        if (count > 4) then
          die
        else
          if randomfloat < 0.3 then
            breed
          else
            if randomfloat < 0.5 then
              move random 4
            end
          end
        end
      end

    end
  end
end
```

LISTING 2.2: A simplistic variant of the Predator-Prey model, written in the SOL DSL (see Chapter 5) for a variant of the Predator-Prey model.

modelling process. This is discussed in section 2.1.5.

2.1.4 Amorphous Computers

To illustrate the interconnectedness of ABM as a methodology and perhaps more as a "mindset" [24], spatial computers, and, more specifically, their *amorphous medium* abstraction [17], present a direct application of multi-agent systems. Amorphous computing is a fascinating concept, where severely limited computing nodes strewn across an area communicate and tolerate communication errors, sensor inaccuracies and even total node failure [39]. While not immediately obvious as an ABM methodology, the simulation of such a system *is* an agent-based model simulation, one of a network-space agent model. Simulating such systems can be a very useful tool to assist in designing them. There are several characteristics in a simulation of an amorphous computer that are related to similar concepts in ABM. Firstly, individual components in the computing medium of nodes are autonomous and operate independently, occasionally communicating with each other locally. Moreover, the nodes are spatially situated.

Researchers at MIT have developed a language and simulator known as Proto [15, 10]. The language itself is LISP-like, and initially intended for prototyping sensor networks. However, with more recent modifications, it is able to simulate actuators as well, in order to simulate robot teams [10]. Proto also supports compiling for actual sensor network nodes which can be deployed.

The effort expended by research teams in this area can, in the most part, be considered when performing ABM. It is possible to implement certain agent-based models in this spatial computing paradigm. Some researchers have already begun to explore the advantages and disadvantages of re-using this research effort in other domains, notably Multi-agent Systems (MAS) and ABM [282, 260].

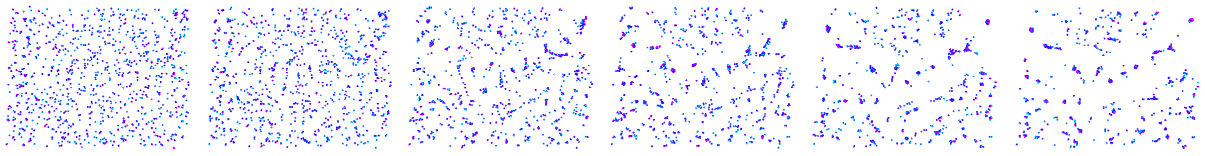


FIGURE 2.6: A time lapse image showing several screenshots of a Particle Swarm Optimiser written in Proto.

Figure 2.6 shows a deliberately slowed Particle Swarm Optimiser (PSO) running in the Proto simulator. This serves as an indication of what Proto is capable of. Proto was even released with an example program of flocking.

Briefly, agents in the simulation of the PSO shown in Figure 2.6 communicate with each other up to a limited distance, and share information about their spatial position. In this case, the absolute coordinates in the system are used as the input variables to a 2-parameter, real-valued objective function. Agents indicate themselves as having the best neighbourhood fitness by switching on a green light, at which point, other nearby agents move towards them with some random variation, in the spirit of the PSO. Optimisation algorithms such as the PSO are discussed in detail in Chapter 3.

2.1.5 Visualisation and Analysis

In order to speed and improve the understanding of a model beyond empirical observation, it is prudent to quantify the simulation of an agent-based model as a whole. This also serves to convey its purpose, which should almost never be simply "realism" [87]. Simple clustering and histogramming (shown in Figure 2.7) are simple techniques to obtain spatial configuration information [115]. Often however, the statistic gathered depends on model purpose, such as in modelling Photobioreactors [95, 125] where accurate growth rate is the objective. The popular modelling platform NetLogo [279] has even enjoyed the addition of a statistical computation facility [278] for this purpose.

Simple spatial clustering and histogramming can be useful, and serves to indicate the usefulness of other tools to obtain information describing spatial configurations. An example of spatial histogramming is shown in Figure 2.7, in which a simulation of Boids in continuous space was executed. The left image shows the 3D space randomly initialised with Boids. Each cuboid in the space indicates the presence of an agent, and its colour indicates the number of agents in that cell. Agents were directed towards the centre of the large 3D cube by an additional "goal" rule. The right image shows the same system several time steps later.

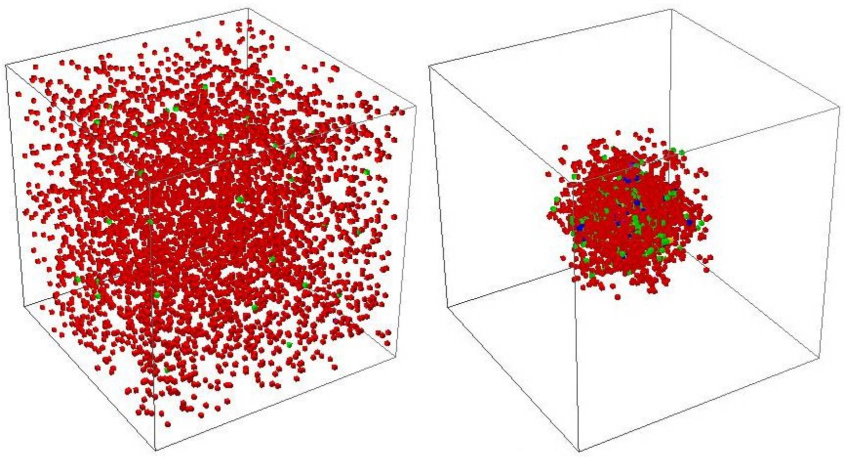


FIGURE 2.7: Visualisation of a sample spatial histogram of a Boids simulation.

Figure 2.8 shows a plot of cluster counts obtained by using a component labelling algorithm on the 3D histogramming data. As the agents tend toward the centre of the cube, the cluster count declines, and tends towards 1. Each curve represents a different number of agents at initialisation. Around frame number 1000, all configurations tended to form 1 cluster. The geometry boundaries were not changed, which caused density to increase linearly with system size, which then caused this effect.

Visualisation is an effective tool for gaining valuable insights in the process of an experiment [106], and specifically with regard to ABM [148]. It is usually somewhat trivial in 2D plane lattice-based simulations and spatial simulations, however, 3D lattices are more difficult to visualise since it is typically only the outer layers that are visible. Programs such as Cubes [93] allows the operator to obtain crossections of a 3D dataset. Figure 2.9 shows different rendering methods of an agent-based photobioreactor model[7]. In this case, coherent clumps of biomass are more important than the cells which are contained within them. The model which produced this image is discussed in detail in Chapter 7.

2.2 Introduction to Parallel ABMS (ℐ, p. 11)

It is well known that processor manufacturers are not increasing clock speeds further, instead, they are adding cores and concentrating on Single-Instruction Multiple-Data (SIMD) architectures [221]. Aside from multiple cores, there are other parallel processors in common desktop machines such as Graphical Processing Units (GPUs).

[7]This is related to the models developed in Chapter 7.

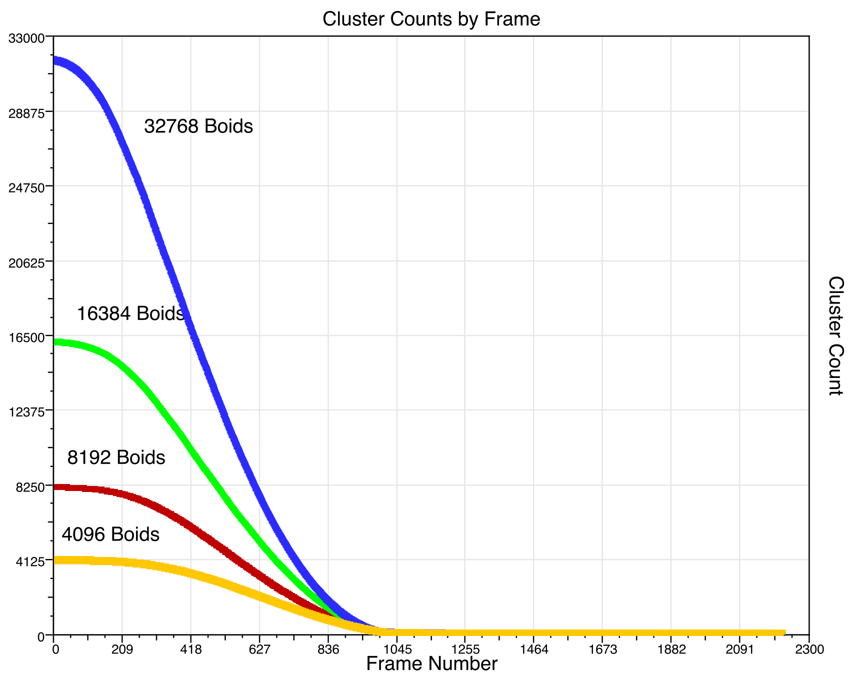
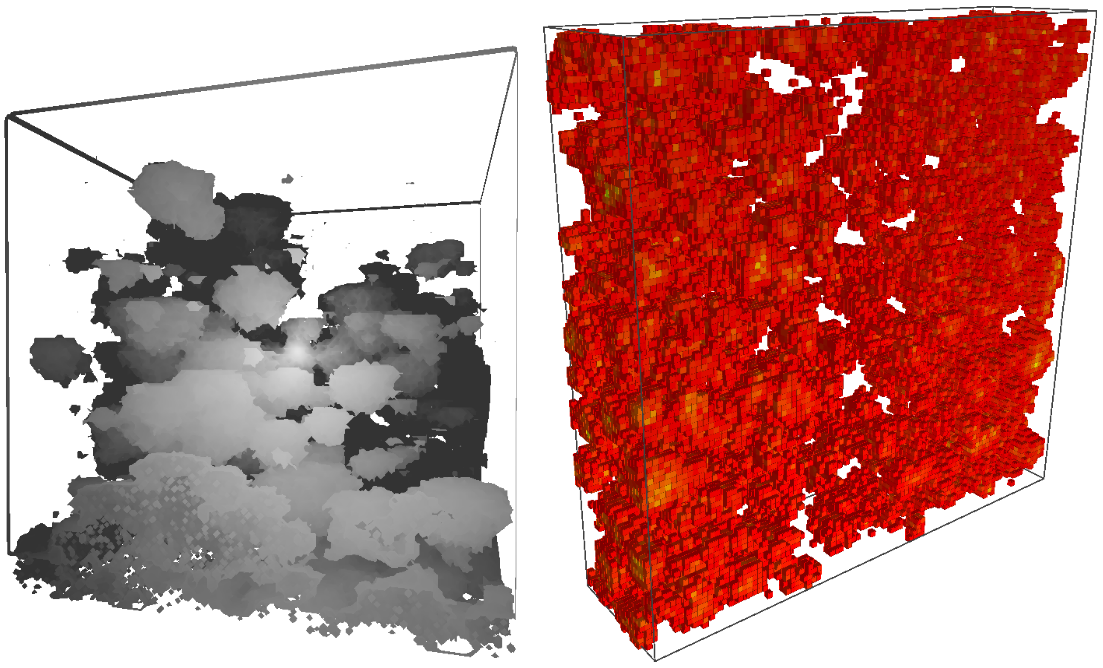


FIGURE 2.8: Cluster counts obtained from a Boids simulation using a simple component labelling algorithm. An additional rule directs all agents toward the centre of a cube, which is why the cluster counts tend towards one.



(a) An isosurface rendering where clumps of biomass are rendered as surfaces using a Marching Cubes algorithm implementation in a superficially modified NVidia CUDA sample program [202].

(b) A point-rendering visualisation method using cubes to indicate cell occupation.

FIGURE 2.9: Example renders of an agent-based photobioreactor model.

GPUs were designed for processing mass numbers of pixel data as quickly as possible, and as such, they have evolved over several years to become massively parallel devices, capable of assigning a single lightweight thread to every pixel. As researchers have discovered decades ago, this pixel data need not necessarily be image data, rather, scientific data.

Several authors have reported the use of GPUs for accelerating simulations, including simulations of agent-based models. For example, particle based simulations [84, 203], Tuberculosis modelling [51], marketing [287] and photobioreactors [125] as well as platforms for accelerating ABM generally [246, 245, 1, 220]. Earlier platforms such as MASON [169] in 2005 supported a maximum of one million agents, provided the visualiser was disabled. But the parallelisation of agent-based models was likely conceived in concept much earlier in 1994, when Mitchel Resnick published "Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds" [239]. Resnick notes on page 45 that while parallelism has received much interest in the 1980s and 1990s, *his* goals (for the educational package "StarLogo") were in fact to support discrete event simulation, where some events must occur simultaneously. Similar concepts and difficulties surfaced in the study of long-standing cellular automata [102], which did result in some research towards parallelisation across, for example, Transputers [30] in 1995. Desktop hardware is perfectly suited to simulating small scale agent-based models, but larger scale models requiring several thousand agents have remained out of reach for some time. This has prompted research by authors such as Scheffer et al., who devised an approximation to several agents known as "super-individuals" [252], which is reminiscent of the concept of a "holon" [76].

More recently, in 2006, Scheutz and Schermerhorn investigated algorithms for automatically parallelising agent-based models across multiple CPUs, thereby saving the ABM researcher the burden of improving performance manually [254]. Despite this advancement, in 2008 researchers at Oak Ridge National Laboratory stated their belief that research in ABM is starting to become stymied by lack of speed and scale; they then introduced the use of GPUs for improving the performance and scaling characteristics of ABM [220], and later, multiple GPUs [1]. In the same year, Lysenko and D'Souza also published a framework for ABM on GPUs [172]. What is clear is that the driving force behind these improvements is a general interest in massive scale and the potential discoveries to be made within.

While the focus here is on the use of fine-grained parallelism using GPUs, it is important to not ignore multi-core processors which are also present in the average desktop computer. Chitty asserts that a carefully written program can extract performance similar to that of GPUs [36] from multi-core CPUs; a provocative statement which in the least forces one not to disregard multi-core CPUs as a method of improving performance. In this dissertation, GPUs are studied for their intensive use later in Multi-Stage Programming and run-time code generation. In this dissertation, performance tuning is necessary to ensure that model structure optimisation efforts in later chapters can be computed in reasonable time. Computing expense in this research is a significant burden.

2.2.1 Compute Unified Device Architecture

NVIDIA's Compute Unified Device Architecture (CUDA) is a potent arrangement of hardware and software extensions designed to streamline general purpose use of NVIDIA's graphics hardware [202]. It makes available a comprehensive back-end library and a compiler (known as nvcc) which filters out special additional syntax to C, and compiles them to CUDA PTX instructions [201]. PTX is the intermediary single static assignment language which is later compiled (as needed) into device specific machine code. The syntax additional to C makes available a mechanism by which to launch code on devices known as "kernels". Additional functions from the CUDA library are also linked automatically to allow copying memory to and from the devices, as well as a host of other functions.

The flowchart shown in Figure 2.10 shows one process which a simulation typically follows when using an NVIDIA GPU.

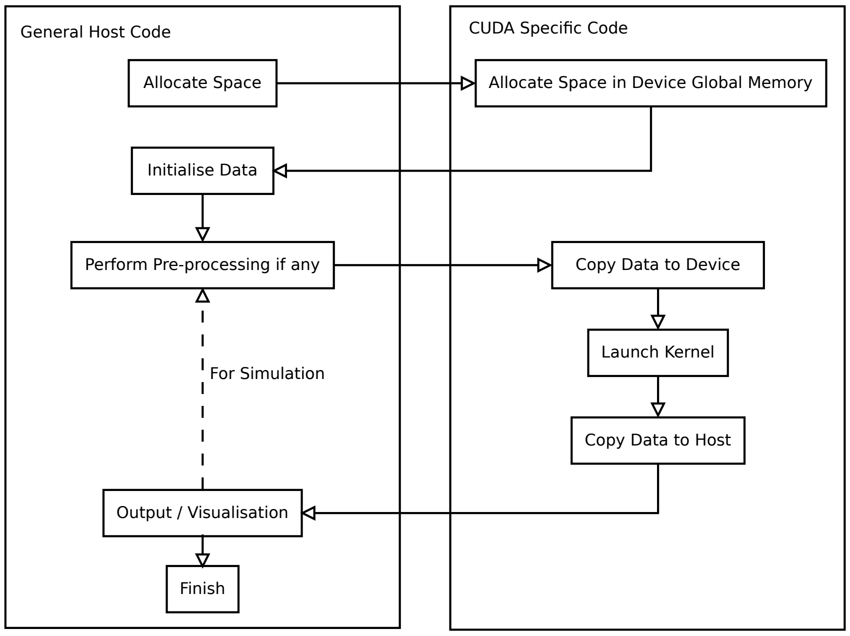


FIGURE 2.10: A flowchart indicating the process that an application typically follows when using an NVIDIA GPU.

This is not the only process, however. It is also possible to use page-locked host memory to map host memory space directly to device memory space, streamlining the data copy across the PCI bus, which is a very time consuming process [202]. To illustrate the process shown in Figure 2.10, a simple minimal program for computing the sum of two integer vectors on an NVIDIA GPU using CUDA is shown in Listing 2.3.

Lines 4–7 defines the CUDA “kernel” which executes directly on the device, and line 28 shows how it is launched using special syntax. These lines are pre-processed by the nvcc compiler before the rest of the code is passed to the system C/C++ compiler. More sophisticated examples are distributed in the CUDA SDK [202].

The numbers between the <<< >>> brackets indicate the thread grid configuration. The thread grid is essentially a hierarchy, determining the number of logical threads to execute. The grid is further subdivided into a series of blocks, each of which can contain up to a maximum of 1024 threads at the time of writing. The grid and the blocks within it may be 1D, 2D, or 3D.

The width, height and depth of the blocks may vary, as can the grid, provided boundaries are not exceeded. These tend to differ depending on the device. CUDA devices are also built with a certain number of Streaming Multiprocessors (SMs), which again vary depending on the device. Each SM may process one block of threads at a time, occasionally switching to another incomplete block in order to hide memory latency. It is important to note that the grid and the blocks within it are not purely hypothetical. Blocks are executed in arbitrary order by different SMs, each of which has a certain number of floating point and double precision units, local memory banks, and shared memory banks, as well as other cache units. Therefore, a single block must adhere to the limitations of a single SM. This may sound daunting, but among all the restrictions and limitations of the CUDA architecture lies the power of a massively powerful processor which, with a well written program, can be effectively harnessed. Complete details and more can be found in the CUDA C Programming Guide [202], key points are extracted here.

```
#include <stdio.h>
#include <assert.h>

__global__ void cuAddVectors(int* a, int* b, int* result, int count) {
   int tx = blockDim.x*blockIdx.x + threadIdx.x;
   if (tx < count) result[tx] = a[tx] + b[tx];
}

int main() {
  const int count = 256*8;
  int* h_a = (int*)malloc(sizeof(int) * count); int* d_a = NULL;
  int* h_b = (int*)malloc(sizeof(int) * count); int* d_b = NULL;

  int* h_results = (int*)malloc(sizeof(int) * count);
  int* d_results = NULL;

  assert(h_a && h_b && h_results);
  assert(cudaMalloc((void**)&d_a,  sizeof(int) * count) == cudaSuccess);
  assert(cudaMalloc((void**)&d_b,  sizeof(int) * count) == cudaSuccess);
  assert(cudaMalloc((void**)&d_results,sizeof(int) * count) == cudaSuccess);

  for (int i=0; i < count; ++i) {  h_a[i] = i; h_b[i] = 1;  }
  assert(cudaMemcpy(d_a,h_a, sizeof(int)*count, cudaMemcpyHostToDevice)
    == cudaSuccess);
  assert(cudaMemcpy(d_b,h_b, sizeof(int)*count, cudaMemcpyHostToDevice)
    == cudaSuccess);

  cuAddVectors<<<256, 8>>>(d_a,d_b, d_results, count); assert(cudaGetLastError()
    == cudaSuccess);

  assert(cudaMemcpy(h_results,d_results, sizeof(int)*count, cudaMemcpyDeviceToHost)
    == cudaSuccess);
  for (int i=0; i < count; ++i) {
    if (h_results[i] != i+1) {
      printf("Failure.\n");
      return -1;
    }
  }

  cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_results);
  delete[] h_a;  delete[] h_b;  delete[] h_results;

  printf("Success\n");
  return 1;
}
```

LISTING 2.3: A minimal program for computing the sum of two integer vectors on an NVIDIA GPU using CUDA 5.0.

A pseudocode kernel for the Boids model is shown in Listing 2.4. This code incorporates a technique known as "tiling" [154, 203, 222] for caching global memory to a faster shared memory bank within the SM a block is executing on. Global memory is notoriously expensive to access, often ranging in hundreds of clock cycles [202]. Moreover, memory accesses occur in fetches of a certain number of bytes, of which most can be wasted by uncoalesced memory accesses. At present, memory accesses are of a fixed size, and the more threads which can complete their memory fetches using the same fixed size memory fetch, the faster memory accesses will be. When threads have completely separate memory fetches in separate parts of memory, this is termed as a lack of coalesced

```
__global__ void boid_kernel(positions,velocities) {
  // compute scalar index from threadIdx and blockIdx
   for (int i=0; i < (N/tile_size); ++i) { // for every tile
      // load tile into memory
      __syncthreads();

      for (int j=0; j < tile_size; ++j) {
       // compute additions to centre of mass calculation
       // compute additions to average velocity calculation
       // compute additions to avoidance calculation
       // increment count of agents in local neighbourhood
      }
      __syncthreads();
   }

   // add centre of mass contribution to my_velocity
   // add average velocity contribution to my_velocity
   // add avoidance calculation to my_velocity

   // compute new position by simple Euler method
   my_position = my_velocity * new_velocity_ratio + my_position

   // check boundaries

   // copy new position and velocity into new device vector
}
```

LISTING 2.4: An interaction CUDA kernel for the Boids model.

reads.

Warp divergence is also an issue: maximum performance tends to be reached when threads execute the same instructions. Threads which are not in the same context as others are simply disabled and essentially execute NO-OP instructions until they are back in context. In some cases, global memory accesses can be “hidden” by the warp scheduler. This occurs when there are sufficient warps ready for instructions while others are waiting for memory fetches.

Some simple performance data was gathered for the Boids simulation, across a range of NVIDIA graphics hardware and a range of system sizes. This is shown in Figure 2.11. As a very rough comparison, the NetLogo (version 5.0.4) Java platform for ABM appears to compute one frame of the included Flocking model with 3000 agents in 0.05s, well above the starting mark of the GPUs.

From a practical standpoint, other matters need to be considered for effectively harnessing the power of GPUs. Apart from “tiling”, it is also necessary to consider how random numbers can be generated on the GPU [161]. Should only one or two random numbers be necessary, it is possible to copy high quality host-generated random deviates onto the global memory of the device, but if any number of random numbers *could* be used, then this will quickly become inefficient and impractical. Various methods are possible including using CURAND (distributed with the CUDA SDK) [97], as well as copying several separate generator internal states to the device. As in all cases, care must be taken in order to ensure a sequence of decorrelated deviates.

It is sometimes necessary to synchronise threads, as in the case of tiling. At first glance, the __syncthreads() internal device function call shown in Listing 2.4 is perhaps misleading. This barrier sync applies to the threads within a block only. By design, it is not possible to synchronise threads across blocks, since the CUDA programming guide states that blocks must be executable independently and in an undefined order [202]. This has far-reaching

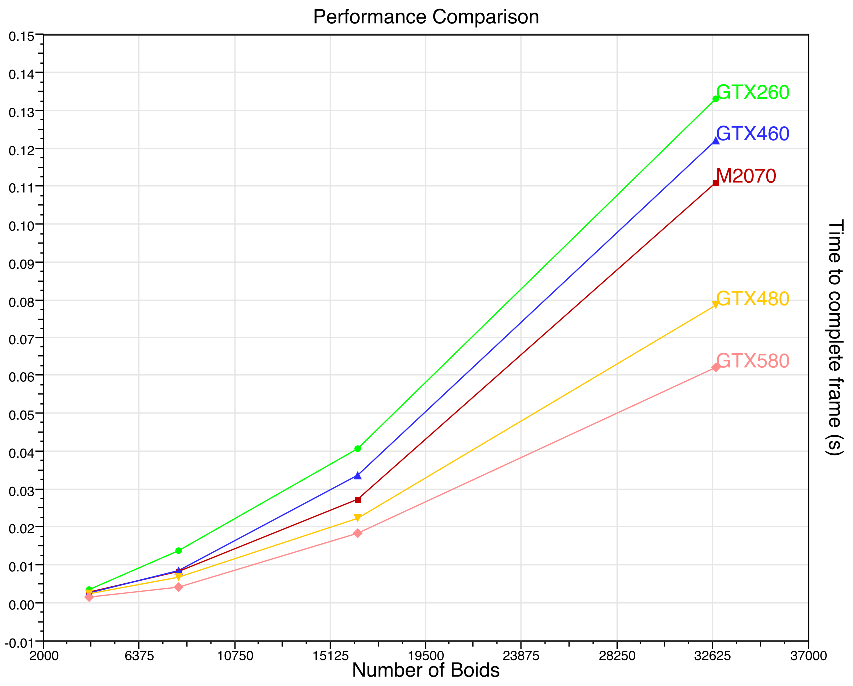


FIGURE 2.11: Performance by system size across a range of NVIDIA graphics hardware.

implications, and must be taken into account during development. It is worth noting that should inter-block synchronisation be necessary, it is possible to synchronise between CUDA kernels when control returns to the host processor.

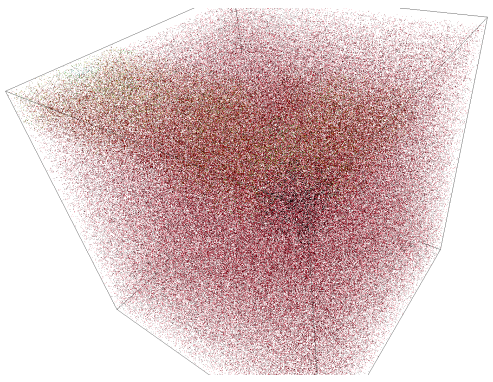


FIGURE 2.12: One million particles running a simple Boids simulation, rendered and computed in real time using CUDA, spatial partitioning and a simple point-rendering pixel shader technique.

Figure 2.12 shows one million Boids, which was rendered in real time. A fast point-rendering method using pixel shaders was used to render the image quickly, and to improve the performance of the algorithm, redundant interactions were removed using a spatial partitioning algorithm [119]. Spatial partitioning for reducing redundant interactions is introduced in the next section.

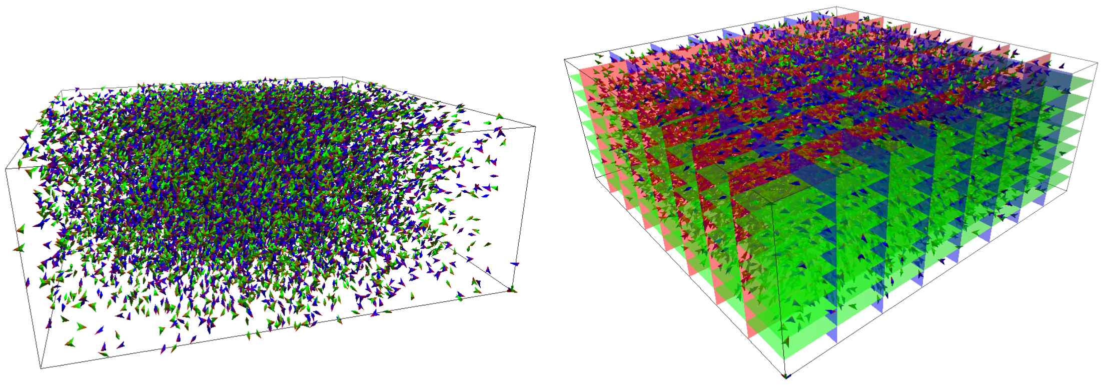
2.2.2 Spatial Partitioning

In cases where a large number of agents are present, and each has a relatively small interaction distance, there is a very large number of redundant interactions taking place. Every agent must effectively communicate with each other, regardless of their spatial location, regardless of whether they will actually interact. In the case of Boids, every agent must iterate over all others and compute a distance in order to determine which agents to interact with. While these are simple communications, the sheer number of these cause a considerable effect on overall performance.

Spatial partitioning [96, 204] was under investigation decades ago, as early as 1985 when Appel proposed the first multipole partitioning method [3] for N-body simulations, with some subsequent developments [265]. The N-body simulation often sees use as a computing benchmark [203, 222]. Other methods of mitigating the so-called "all-pairs" issue surfaced including the Barnes-Hut method [12], tree organisational methods [290] including Octrees [291], K-D trees [44, 297] (shown in Figure 2.14), Kautz trees [303], Adaptive Refined Trees [152], and more. K-D trees are especially popular for ray tracing applications [44]. Some of these are suitable for particle-based simulations, where high density is not an issue, and others perform best when agents or particles are approximately uniformly distributed across space, and some are approximations trading some precision for speed by treating dense space as point masses (particularly of interest in Astrophysics), similar to the ideas of Scheffer [252], although for a different purpose. The choice of spatial partitioning algorithm therefore certainly depends on the purpose of the simulation.

A problem domain which had a particular role in the development of these techniques is collision detection in computer science [204]. Another $\mathcal{O}(n^2)$ complexity issue. While this problem has great importance in simulations of physical objects, these ideas are also applicable in several other domains [204]. In the case of this work, interactive agent-based simulations.

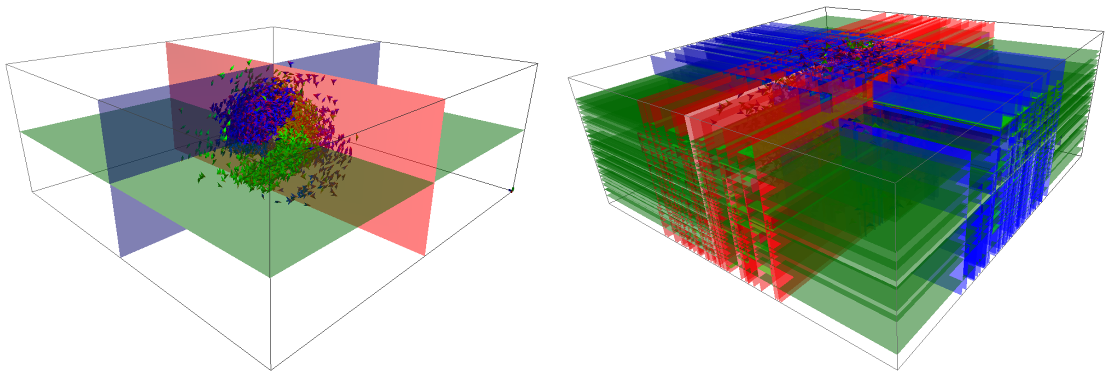
The uniform grid partitioning method [61, 96] is a particularly simple algorithm among other spatial partitioning algorithms, but its simplicity lends itself well to exploitation on GPU [84], both during construction of the datastructure, as well as the actual use of it for lookup procedures. This is fortunate considering that certainly not all space partitioning algorithms respond well to parallelisation [204]. The rest of this chapter makes use of this conceptually simple algorithm to improve system scaling characteristics of the Boids model.



(a) Screenshot of the Boids simulation.

(b) Boids with a uniform grid visualisation superimposed upon it.

FIGURE 2.13: Flocking: Visualisation of a uniform grid partitioning scheme over a set of agents.



(a) First 3 axis splits of the kD tree. (b) Full kD tree.

FIGURE 2.14: Flocking: Visualisation of simple and complex constructed k-D trees.

Even more fortunate is the fact that the uniform grid datastructure can be constructed on GPU, where it would be used, as Green demonstrated elegantly [84]. A GPU-based implementation of this is shown in Algorithm 2 which is heavily based on the implementation of Green's particle simulation, but adapted for Boids [84]. The way in which space is partitioned is straightforward, and shown in Figure 2.13.

The algorithm shown in Algorithm 2 is very similar to the method which Green used to accelerate a particle simulation demonstrating collision detection using CUDA [84]. In his simulation, collision detection ensured that spatial density was not an issue. As seen in the algorithm shown for Boids, most computations are done in parallel, such as hash computations, sorts by hash key, populating boxStart and boxEnd with indices of present agents, scatter writing all boids to their hash-key array location, and finally, computing interactions using another CUDA kernel.

In the main loop of the algorithm, hashes are first computed for all agents. The hashes are simply Morton codes in order to accomplish Morton ordering. Morton ordering (or Z-ordering) is a space-filling curve used to ensure that stored data reflects spatial locality [176]. In this case, the Morton codes are computed by the spatial location of the agent in a uniformly divided grid. This method helps greatly to reduce costly additional memory fetches in CUDA.

The indices array is simply filled with a zero and the sequence of positive integers. This is so that the sequence of hashes can be sorted while still keeping track of which agents they refer to. After the hashes and indices arrays have been sorted by hashes, then another CUDA kernel populates an array of grid box starting and ending indices. These arrays are used by the interaction computing CUDA kernel. By iterating over agents beginning from the start to the end of the box indices provided, many redundant interactions can be eliminated. Of course, agents bordering the individual cells could possibly suffer the opposite issue: a lack of communication with an agent clearly within radius. This problem is solved by checking surrounding grid boxes as well. Due to this, it is very important to choose a grid granularity which suits the communication radius of the agents. The size of an individual grid cell should at least be large enough to completely contain the communication radius of an agent. Much larger than this, and the datastructure loses its effectiveness; much smaller than this and communication errors will be introduced, or too many grid boxes must be checked.

There is a considerable amount of computing taking place in Algorithm 2, and yet this still improves performance over simple implementations of simulations with an inherent $\mathcal{O}(n^2)$ complexity. Further performance improvements can be obtained by making use of multiple GPUs in a single computer, especially since high-end NVIDIA graphics cards often have two physical GPUs integrated within. This is introduced in the next section.

```
Allocate memory space
copyVectorsToDevice()
for $i \leftarrow 0$ to $n$ timesteps OR NOT exit_condition do
  $i \leftarrow i + 1$
  $\{$Calculate hashes for all boids.$\}$
  for $j \leftarrow 0$ to $NUM\_BOIDS$ do
    hashes[j] = calculate_morton_code(position[j])
    indices[j] = j
  end for
  Sort by hash key (hashes,indices)
  Populate boxStart and boxEnd
  Scatter write boids
  for each agent in parallel do
    for every adjacent grid cell $0 \leq g < 8$ do
      for every agent $a$ in grid cell $g$ do
        if $a$ in communication radius then
          Compute rule contributions
        end if
      end for
    end for
    Update position and velocity using averaged contributions
  end for
  copyVectorsFromDevice()
  drawBoids()
  swapDeviceBuffers()
end for
```

ALGORITHM 2: GPU implementation of a uniform grid spatial partitioning scheme for the Boids model.

2.2.3 Multiple GPUs

Aaby and colleagues at Oak Ridge National Laboratory have shown that multiple GPUs can be harnessed together in order to improve performance further [1]. While there is a considerable performance increase in using multiple GPUs, the underlying complexity is not mitigated at all without using some method such as spatial partitioning.

Essentially the parallelisation process depends upon what strategy is used. In the case of ABM, the most immediate method would be to assign a thread to every agent. This is “fine-grained” parallelism. Unless each agent has independent tasks which can be parallelised, there is no other logical way to assign more threads to the problem. “Coarse-grained” parallelism in an agent-based model would be assigning one thread (or processor) to several agents.

By using the uniform grid developed above, one can further improve performance of a multiple-GPU agent-based model. Algorithm 3 shows a simple method to accomplish this. Note that it is not necessary for all devices to build the datastructures independently. However, since a timestep cannot be computed before this happens, every device computes the datastructure separately. This avoids having to distribute the datastructure to every device, which would incur an additional memory copy penalty.

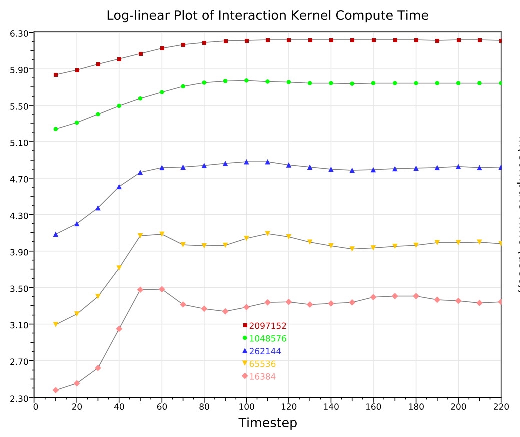
```
Allocate arrays
copyVectorsToDevices()
for $i \leftarrow 0$ to $n$ OR NOT exit_condition do
  $i \leftarrow i + 1$
  for each GPU in parallel do
    for $j \leftarrow 0$ to $NUM\_BOIDS$ in parallel do
      hashes[j] = calculate_hash(position [ j ])
    end for
    Sort by hash key (hashes,indices)
    Populate boxStart and boxEnd
    Write agents to their sorted locations
    for each agent in parallel do
      for every adjacent grid cell $0 \leq g < 8$ do
        for every agent $a$ in grid cell $g$ do
          if $a$ in communication radius then
            Compute rule contributions
          end if
        end for
      end for
      Update position and velocity using averaged contributions
    end for
  end for
  copyPartialVectorsFromDevices()
  copyVectorsToDevices()
  drawBoids()
end for
```

ALGORITHM 3: Multiple-GPU (mGPU) implementation of the uniform grid in Boids.

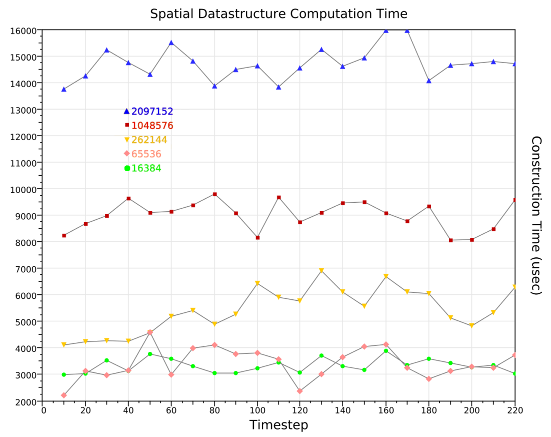
In Algorithm 3, a timestep is computed by first copying velocities and position vectors to all devices. Identical information is cloned onto every device. It is not necessary that all devices are aware of all agents. It is sometimes suitable to distribute distinct parts of the space (either lattice or continuous space) to the devices with a read-only border [1]. This is one possible enhancement to the above algorithm. After the GPUs have constructed the uniform grid, they compute the CUDA kernel, which uses it by iterating over each adjacent grid cell, and the agents within them. Finally, the agents are updated, and the host copies back the sequences of agents modified by each GPU. The process repeats when the host reconstructs the full arrays once more and copies these to the devices. There are, however, some redundancies here which can be eliminated.

The most immediate drawback here is that devices must synchronise in order to construct the whole array of agents, which is fed back to the GPUs again. This can be improved using newer versions of CUDA released, allowing several devices to communicate with each other directly using a unified address space [202]. Historical releases of CUDA have improved upon each other significantly, and the trend seems to continue. Page-locked memory can also be used to construct the full array of agents, which will also improve performance.

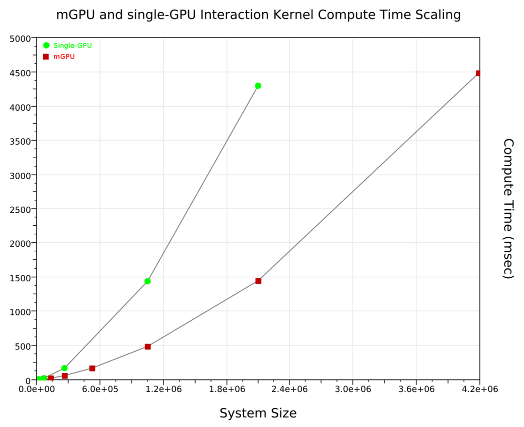
It is also very important to consider the memory hierarchy at almost every step of development in these programs. Memory fetches from global memory is extremely slow in comparison to memory fetches from the registers and shared memory on each SM of the GPU. Also, the constant memory bank should not be ignored, which is a cached segment of global memory. Use of the texture cache can also be a source of improvement here, since it is designed to operate at maximum efficiency for an algorithm which is likely to benefit from spatial locality in its data [202].



(a) A log-linear plot of agent-agent interaction time step compute time using the CUDA kernel with different system sizes across timesteps.



(b) Datastructure construction time plot by system size for mGPU.



(c) Comparison of mGPU and single-GPU algorithms for timestep computing by system size.

FIGURE 2.15: Multiple-GPU performance results for datastructure construction and timestep computation.

Figure 2.15 shows some performance data for the multiple-GPU implementation. As expected, the multiple-GPU algorithm increases in compute time more slowly than the single-GPU algorithm (Figure 2.15(c)). Although there could be some algorithmic improvements, a slower increase in compute time is certainly desirable. Constructing the spatial datastructures appear to be fairly consistent for different system sizes using the mGPU algorithm (Figure 2.15(b)). Figure 2.15(a) shows that the first few timesteps of a simulation run increases in computing time quickly, followed by a more consistent increase in computing time. The reason why computing times increase consistently is due to the agents using less grid boxes while moving towards the centre of the space. Ideally, spatial distribution of agents would be more uniform for maximum utility from the spatial partitioning algorithm.

Part II

Parallel Evolutionary Algorithms

CHAPTER 3
CONTINUOUS GLOBAL OPTIMISATION

Many problems can be described in terms of a real-valued cost function. For example, a wind turbine blade requires a design which maximises *output power* given specific weather conditions [138]. Hypothetically, the output power $P = F(f, r, m)$ could be expressed in terms of fibreglass thickness f, number of ribs r, and perhaps more directly on mass m. Furthermore, these parameters may be subject to an upper and lower limit due to tolerances in allowable blade stresses and displacements. Optimisation of cost functions such as these is relatively simple, but the choice of algorithms is vast enough to cast doubt on "off-the-shelf" solutions, especially given recent theoretical advances. Such optimisers are broadly categorised into *population*-based, and *trajectory*-based methods. Some are then further divided into stochastic and deterministic. Problems themselves can also be categorised into constrained and unconstrained.

The purpose of this chapter is to introduce and characterise a selection of major stochastic numerical optimisers in terms of data-parallel computing, applicability in calibrating agent-based models, and for use later in parallel geometric optimisation in Chapter 4. Optimiser evaluation methods and some notable variations in algorithm design are discussed. Spatial partitioning concepts are adopted from Chapter 2 in order to propose improvements to these algorithms. Advanced space exploration techniques such as Lévy Flights and variations of these are discussed. Finally, a study on optimiser calibration is presented. Given the fundamental similarities between population-based optimisers such as the PSO and agent-based models, calibration of these can be equated, from an optimisation point of view. The chapter ends with a short study on higher dimension visualisation.

The contents of this chapter extend upon work previously published by the author in *Proc. 12th IASTED Int. Conf. on Artificial Intelligence and Applications*[1], *Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC 2013)*[2], *Parallel and Cloud Computing*[3] and also *Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM 2012)*[4].

[1]A. V. Husselmann and K. A. Hawick. Random flights for particle swarm optimisers. In *Proc. 12th IASTED Int. Conf. on Artificial Intelligence and Applications*, Innsbruck, Austria, 11-13 February 2013. IASTED

[2]A. V. Husselmann and K. A. Hawick. Particle swarm-based meta-optimising on graphical processing units. In *Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC 2013)*, Phuket, Thailand, 10-12 April 2013. IASTED

[3]A. V. Husselmann and K. A. Hawick. Levy flights for particle swarm optimisation algorithms on graphical processing units. *Parallel and Cloud Computing*, 2(2):32–40, April 2013

[4]A. V. Husselmann and K. A. Hawick. Parallel parametric optimisation with firefly algorithms on graphical processing units. In *Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM'12)*, number 141 in CSTN, pages 77–83, Las Vegas, USA, 16-19 July 2012. CSREA

3.1 Introduction

CONTINUOUS OPTIMISATION ALGORITHMS ADDRESS MANY PROBLEMS in a variety of areas such as image compression [109, 110], improvement of manufacturing processes [5], structural design [72, 9], scheduling problems (notably, Job-shop scheduling problems) [143], cryptanalysis [208], object recognition and clustering [259], economics (notably the Load Dispatch problem) [302, 79, 2], antenna design [34, 14], spring-mass systems [53] and more. Each of these problem domains do not necessarily use the same set of algorithms. There are many to choose from, and some are more suitable than others for certain tasks.

The field of numerical optimisation in general is vast, and includes a wealth of gradient-based methods, stochastic algorithms such as evolutionary optimisers, and many hybridised methods. In general, the problem of optimising a function with real-valued parameters is defined by Nocedal and Wright [199] as:

$$\min_{x \in \mathbb{R}^n} f(x) \begin{cases} c_i(x) = 0 & i \in \mathcal{E} \\ c_i(x) \geqslant 0 & i \in \mathcal{I} \end{cases} \tag{3.1}$$

More specifically, this is the minimisation of a function with an input vector x in $\mathbb{R}^n$, subject to a set of constraints on each variable. $\mathcal{E}$ and $\mathcal{I}$ are simply integer sets containing the indices of constraint functions. Of course, in the case where $\mathcal{E} = \emptyset$ and $\mathcal{I} = \emptyset$, the problem is *unconstrained*, which, while a special case, is a considerably different problem to solve. The methods to solve constrained and unconstrained optimisation problems are quite varied [292].

Focus in this chapter is given to *global* optimisation (which is contained within numerical optimisation [292]) and deals with the task of finding a *global* optimum for a certain known or unknown function $f(x)$ across the entire set of $\mathbb{R}^n$. The search for a global optimum is made more difficult by the presence of local minima and local maxima. These are simply the best solutions for a specific *subset* of the input parameters (solutions which are the best for a *local* region) but not the whole set of input parameters. These tend to cause algorithms to converge, in the same manner as it would for a global optimum.

Specific interest here is given to *stochastic* optimisation using Evolutionary Algorithms (EAs). EAs are attractive due to their inherent parallel natures, which can be exploited effectively in the same style as agent-based models. The greater subset of stochastic optimisers are sometimes known as metaheuristics [168] in recognition of the few assumptions made by these algorithms of the target problem. As demonstrated by Section 3.5, this is also attractive because gradient information may be entirely missing.

The problem of finding the real-valued vector x which minimises $f(x)$ has been successfully addressed by various derivative-free optimisers such as Genetic Algorithms [107], Simulated Annealing [145], Particle Swarm Optimisation [142, 262], Firefly Algorithm [299], and other bio-inspired algorithms [298], as well as first-order optimisers such as Gradient Descent and its many variants [199]. It is important, however, not to disregard gradient-based (first-order) methods due to their use of derivatives. The value in using a derivative-free optimiser is the ability to treat a function as a "black-box", which is particularly useful when gradient information is unavailable. For context, a brief introduction on first-order optimisation is provided here.

3.1.1 First-order Optimisation

Algorithms such as Gradient Descent, Quasi-Newton and other variations termed gradient-based methods [199] are known as first-order optimisation algorithms due to their use of first-order derivatives. Assuming that gradient information is available, these algorithms typically follow a deterministic process to iteratively improve upon a solution vector using gradient information.

Gradient descent involves moving from a random location on a curve towards the global optimum by successive additions of $0 < \alpha < 1$ step-sized instantaneous gradients [168]. A simple example is an unknown function $f(x)$, with a known derivative function $f\prime(x)$. Assuming that there are no bounds on the values of x (ie. an *unconstrained* problem), x can be initialised to an arbitrary location (such as $x = 0$) and then $f\prime(x)$ can be iteratively added to it until $f\prime(x) = 0$. At this point, the global optimum would have been found, assuming the function does not contain local minima (monotonically decreasing).

In stochastic optimisation, some randomisation is employed either by perturbing a single solution in space and accepting a better solution with a certain probability (such as Simulated Annealing [145]), or in a population of individuals where collective cooperation tends to accept better solutions with some inertia (perhaps better described as "scepticism").

3.1.2 Stochastic Derivative-free Optimisation

Nocedal and Wright note that should gradient information not be available, it is often adequate to obtain an estimate using finite differencing [199]. They do concede that this is not always appropriate when there is the potential for noise. Generally, algorithms which do not rely on derivatives make very little assumptions about the problem at hand. These algorithms treat the optimisation function as a "black box" [168]. One or more candidate solution vectors are improved by using its corresponding value of optimisation function as a measure of "fitness". It may also be computationally expensive to compute this fitness, which further demands that the optimiser make as few iterations as possible to minimise the number of evaluations performed. This is sometimes used as an additional measure of the effectiveness of an optimiser [299].

Derivative-free optimisers are categorised into those which are based on trajectory methods, and population-based methods [168]. Trajectory-based methods involve the use of a single candidate solution which is improved over time. Examples of these include the Hill-Climbing technique, which is a very simple technique conceptually similar to Gradient Descent. A more sophisticated example is Nelder and Mead's Simplex method [196] and Kirkpatrick's Simulated Annealing [145] algorithm. These methods evolve a certain candidate solution over time, the former being a generalised simplex, and the latter a point in n-dimensional space when applied to continuous optimisation problems.

Elegant nature-inspired algorithms such as the Firefly Algorithm (FA) [299] and the Particle Swarm Optimiser (PSO) [142] make use of firefly flashing behaviour and bird flocking behaviour respectively. While not strictly constrained to their natural counterparts, they still show their source of inspiration in their formulation. The FA for instance, uses a light decay function to degrade the perceived fitness of other fireflies based on the distance between them. The PSO departs somewhat from its natural source of inspiration, but still maintains either a global or local "flock leader", depending on the variant (*gbest* or *lbest*). These algorithms are unified by the term Evolutionary Algorithms, which accentuates their inspiration in some form from simplified natural phenomena [21] rather than specific usage of evolutionary and genetic phenomena.

There are several problems which plague the field of evolutionary algorithms, which are given with regard to the fitness landscape of the optimisation function by Weise and colleagues as [293]:

1. *Deceptiveness* - Areas where gradient information is misleading.
2. *Neutrality* - A zero-gradient in ranges of the optimisation function.
3. *Epistasis* - Interdependency among parameters with respect to the objective function value.
4. *Premature Convergence* - A stagnation of the search algorithm before the global optimum is found.
5. *Ruggedness* - High variation in gradient information causing a "rugged" fitness landscape.
6. *Noise* - The presence of inaccuracies or stochasticity.

A function is *deceptive* should it contain local optima. If it contains regions of zero or near-zero gradient, it is said to contain *neutrality*. *Epistasis* occurs when a function's variables are dependent on each other in some fashion. *Premature convergence* occurs when an optimiser's search stagnates around a local optimum, rather than a global optimum. A function is *rugged* if there is great variation in gradient data, and finally, a function may also be *noisy* if it is subject to Gaussian noise, or simply inaccuracies brought on by factors such as stochasticity.

These problems pertain to the function's landscape. Weise et al. also discuss additional problems in the *algorithms* themselves, including overfitting and over-simplification [293]. Given all these difficulties, a number of algorithms have surfaced which perform differently in the presence of each of these. For instance, it has been suggested that a particle swarm optimiser adapted for use in combinatorial search spaces may be generally less effective than Genetic Programming [280].

Using the "No Free Lunch" theorems of Wolpert and Macready [294], it can be shown that the number of specialised EAs will increase [293], due to the impossibility of one unifying EA outperforming every other EA on every problem. Restricting the domain of an EA allows one to tailor an algorithm to specifically overcome the issues listed above to some extent. This argument was used by van den Bergh, who made the assumption that an algorithm can be designed to successfully outperform others in a specific subset of problems [285].

To be able to compare algorithms by how well they solve optimisation problems in the presence of specific issues is very important. Their stochastic nature also ensures that not every execution is precisely the same. It is therefore sometimes difficult to objectively compare these algorithms, but fortunately, there is a variety of test functions that have been designed for this purpose. As will be explained later, there are other means of comparison. The focus of this chapter is on the use of evolutionary algorithms, due to their ability to optimise functions which have no obvious or accessible analytical forms. Such functions may simply be the measurement of some quantity within a simulation of a fully constructed agent-based model.

3.1.3 Calibrating Agent-based Models ($\mathcal{I}$, p. 11)

As mentioned in Section 1.6, calibrating the parameters of an agent-based model can be seen as an optimisation problem. A notable example of this was investigated by Calvez and Hutzler who used a Genetic Algorithm to optimise various aspects of the ant foraging model [28]. In that paper, the authors bring forth three key issues specifically relevant to this practice:

1. The choice of fitness function.
2. Random variation in fitness due to model stochasticity.
3. Computational cost.

It must be at least possible for an optimiser to compute the fitness of a parameter vector. Precisely how this is done depends on the goal of the optimisation effort. In general, it is either the detection of a desired phenomenon in the system (quantified by some method), or simply the collection of a specific quantity. Obtaining an output value from the simulation of a model with a particular parameter set is effectively the "black box" objective function mentioned above. Clearly, in this case, gradient information is unavailable, and estimation is very difficult.

Random variation stymies comparison between parameter sets. It is then necessary to obtain averages or modify the optimiser, depending on precisely how much variation the simulation exhibits. The issues given by Weise et al. above [293] subsume random variation as a difficulty experienced by stochastic optimisation in general, but describes other problems which are also relevant, but perhaps not as critical.

It is important to consider computational cost as well. For instance, assume some quantity is gathered from timestep 200 of a simulation of ant foraging, this can be seen as a single evaluation of a fitness function. Given a set of 50 different ant foraging models, this would equate to 10000 timestep computations. To reduce variations in fitness, it may also be necessary to average this result 10 times. This is only a single set of 50 candidates, and there may be hundreds depending on the complexity of the fitness function. It is clear that this kind of problem is very computationally expensive. Calvez and Hutzler [28] note that these fitness evaluations are independent, and could be done across a cluster computer, each with a certain number of candidate solutions. This is one method of reducing the time taken by such an optimisation effort.

In the next few sections, the concept of a particle-based optimiser and an evolutionary optimiser is introduced, along with the methods used to evaluate their efficacy. Focus is given to particle-based methods and evolutionary algorithms as these are (1) well-suited to the problem of calibrating an agent-based model, (2) are conceptually simple, and (3) parallelise readily. Specific examples of these optimisers are given in Section 3.2. Then in Section 3.3 advanced space exploration techniques are discussed. Parallel implementation of a selection of these algorithms are presented in Section 3.4 along with algorithmic modifications. The problem of calibrating an optimiser is discussed and mitigated to some extent in Section 3.5. Visualisation of the n-dimensional particle systems is considered in Section 3.6.

3.2 Evolutionary Optimisers

Evolutionary Computation (EC) consists of a set of algorithms which include EAs, and has in recent years become the "umbrella term" used for the majority of population-based optimisation algorithms [267]. EAs iteratively improve upon a population of candidate solutions using population operators inspired by nature, such as mutation and crossover [267, 63] but also linear combinations. Popular evolutionary algorithms include the Genetic Algorithms (GAs) [107], PSOs [142], FAs [299], and geometric versions of these [189]. Some of these may also be classified as metaheuristics [168, 298].

Even though these methods can be considered non-deterministic, they are not by any means less precise, however. The method by which they explore solution space is stochastic, but carefully formulated in order to ensure a consistent convergence to an optimum. This is generally done by exploiting adequate solutions in tandem with some random steps in an attempt to improve them. Any solution found is valid, even when the objective function is constrained, as these algorithms typically constrain their individual candidates to the corresponding hypercube in solution space where each n-vector is a valid solution. There is, however, a risk that the global optimum is not found, possibly due to one or more of the issues discussed in the previous section. In this case, the operator can either use the solution if it is adequate, or modify parameters and try again. The act of modifying optimiser parameters is a dubious practice, but often necessary. This is discussed in some detail in Section 3.5.

The PSO [142] is part of a larger family of particle-based methods, where a set of particles in $\mathbb{R}^n$ pseudospace cooperate to some extent in order to find the global optimum. Frequently, they are actually constituted by both deterministic and stochastic components, as is the case in the PSO and the FA. These facilitate what is known as the solution space exploration component and the discovered solution exploitation component.

There have been a plethora of EAs proposed for global optimisation problems. Searching through these for an algorithm which performs satisfactorily in a particular domain is difficult. One is then generally also required to perform hand-calibration. Fortunately, for the purpose of evaluating algorithms, many test functions exist which take various forms [301, 188]. These are generalised to n dimensions because many optimisation problems are themselves multi-dimensional. The example of an aerofoil design for instance, requires several tuned parameters. The efficacy of continuous global optimisers and metaheuristics are evaluated by using one of these test functions as a "black box".

The search for an appropriate test function is difficult and been under scrutiny for several years. The main reason for this is that finding a suitable test scenario which is representative of all possible problems is an unrealistic goal. Research towards this has, however, resulted in a great variety of functions, where each of which attempts to deceive an optimiser in one or more ways [188]. Comparing against a selection of these provides an acceptable indication of effectiveness with respect to the specific difficulties of the functions and provides for a method by which to objectively compare algorithms.

An example of a test function is the Rosenbrock function [248]. Initially, it was intended for two dimensions, and became known for its characteristic valley, which occasionally earns it the title "Rosenbrock's Valley". A plot of the 2-parameter Rosenbrock function is shown in Figure 3.1(b); the function is shown in Equation 3.2 and its n-dimensional version is shown in Equation 3.3 [188]. The valley in the function is arguably what makes this function difficult to optimise. For an optimiser to locate the valley is not difficult, but to converge towards the global minimum (at $x = y = 1$ for the 2-dimensional case) within it takes many steps. A near-zero gradient in the valley pertains to neutrality, as described by Weise et al. [293]. Other test functions used in this chapter are shown in Figure 3.1.

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{3.2}$$

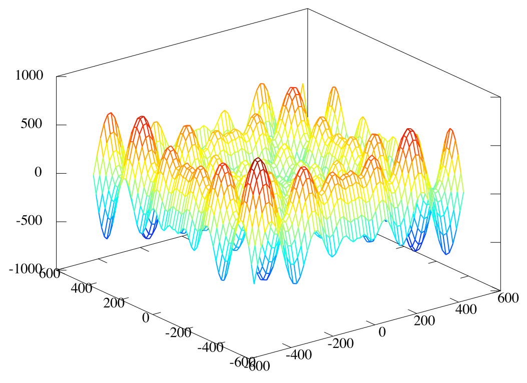
$$f(x) = \sum_{i=0}^{n-2} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \tag{3.3}$$

Each of these functions attempt to deceive the optimiser in a number of ways. Functions such as the Schwefel function [188], for example, distributes local minima throughout the parameter space, geometrically distant from each other and the global minimum. Its n-dimensional version is:

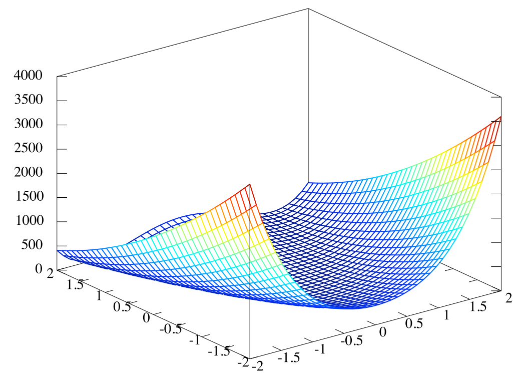
$$f(x) = \sum_{i=1}^{n} \left[- x_i \sin(\sqrt{|x_i|}) \right] \tag{3.4}$$

Other functions such as the Ackley function (shown below) is a more computationally expensive function to evaluate. In cases where the function is excessively compute intensive, it is sometimes useful to measure the number of function evaluations an algorithm computes [299].

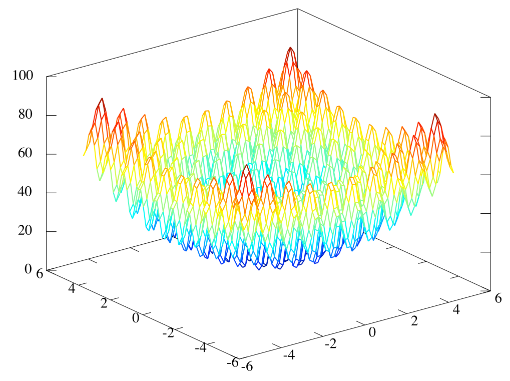
$$f(x, y) = -20 \exp(-\frac{1}{5}\sqrt{\frac{1}{2}(x^2 + y^2)}) - \exp(\frac{1}{2}(\cos(2\pi x) + \cos(2\pi y))) + 20 + e \tag{3.5}$$



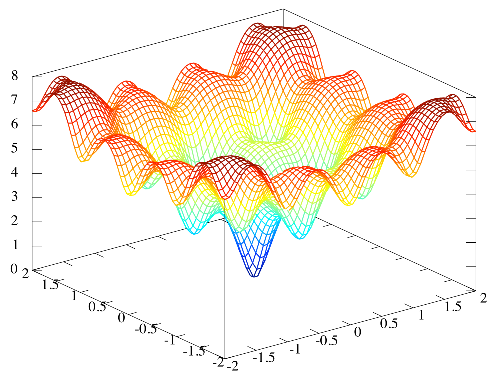
(a) Schwefel function (deceptive and rugged) $f(x, y) = -x \sin(\sqrt(|x|)) - y \sin(\sqrt(|y|))$.



(b) Rosenbrock function (high degree of neutrality and epistasis) $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$.



(c) Rastrigin function (deceptive and rugged) $f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y)$.



(d) Ackley's function (deceptive) $f(x, y) = -20\exp(-\frac{1}{5}\sqrt{\frac{1}{2}(x^2 + y^2)}) - \exp(\frac{1}{2}(\cos(2\pi x) + \cos(2\pi y))) + 20 + e$.

FIGURE 3.1: Surface plots of the 2-parameter variants of The Schwefel, Rastrigin, Ackley and Rosenbrock functions [188].

Boundaries and minima for the test functions used are given in Table 3.1. The lower and upper limits pertain to every dimension of the test problem, and the number of dimensions for each test function is noted when they are used. The minimum of the Schwefel function is -3351.8632, but for convenience, this was renormalised to zero. In all experiments conducted, the functions were minimised.

	Lower	Upper	Optimum
Rosenbrock	-2	2	0.0
Rastrigin	-5	5	0.0
Schwefel	-500	500	0.0
Ackley	-20	20	0.0
Griewangk	-600	600	0.0
Michalewicz	0	$\pi/2$	-4.687

TABLE 3.1: Test functions used and the constraints designating the feasible region of the solution space for each [301]. These bounds are enforced for all dimensions of the corresponding function. In all cases, the optimum is taken to be the global minimum of the function.

The rest of this section introduces prominent evolutionary algorithms including the Genetic Algorithm, Particle Swarm Optimiser, and the Firefly Algorithm. As it is impossible to enumerate all the evolutionary computation techniques for continuous-space optimisation problems, only this small selection of algorithms will be explained in detail, the GA for its conceptual applicability in Chapter 4 and being one of the oldest EAs in history, the PSO for its performance characteristics in terms of parallelisation, and the Firefly Algorithm for its ability to solve optimisation problems in very few time steps.

3.2.1 Genetic Algorithm

A very prominent nature-inspired numerical optimiser is the GA, due to Holland in 1975 [107]. It was hailed as a breakthrough, and the number of variations [82] and improvements since its inception has resulted in successful applications in several disciplines. For instance, Jim and Giles used it to show that communicating predators in a Predator-prey model [257] results in better performance in catching prey than predators that do not communicate [133], a discovery which has important implications. Jureczko et al. used a modified GA to optimise wing structures [138] for wind turbines. Koza used the GA as inspiration for Genetic Programming (GP), a combinatorial optimiser which searches through the pseudospace of programs which also attempts to optimise a quantitative measure of fitness.

The GA is an evolutionary algorithm which maintains a population of candidate solutions (sometimes known as chromosomes), each traditionally in the form of a bit string, though formulation varies. Each candidate could be composed of a predetermined number of symbols, which could be integers, reals or traditionally bits. What makes the GA different from other stochastic optimisers such as Simulated Annealing [145] is its use of a set of operators inspired by nature which determine how to construct a new, better population of candidates. These operators are in the simplest case: selection, mutation and crossover, in the spirit of evolution. Since Holland introduced the GA, many variations have surfaced. Formulations differ in subtle ways, but conform to at least the three above. Without mutation, only genetic drift is possible. Without crossover, candidates cannot exchange information, and the optimiser degenerates to a slow random search. Selection provides meaningful inputs to the crossover operator, without it, fitness becomes meaningless, as the search would not be biased towards better candidates.

The process a GA follows is to first initialise a population of solution candidates p_0 with arbitrary information. This population is evaluated, and each candidate assigned a score by how well they solve a particular problem. A candidate therefore encodes all information necessary to encapsulate a solution to a particular problem. A selection mechanism such as a *roulette wheel* then chooses a set of candidates with replacement. The roulette-wheel *selection* mechanism simply selects candidates with a probability proportional to their fitness. The *crossover* operator then takes effect with probability $P(\text{crossover})$, which takes as input 2 candidates and produces two new candidates by some form of recombination. The *mutation* operator takes effect with probability $P(\text{mutate})$ and takes as input one candidate, and perturbs a small portion of it. The result from these three operators is a new population the same size as the previous one. This process then repeats until a suitable solution is found within a given tolerance, or a maximum number of iterations are computed.

As mentioned previously, many different formulations and variations exist, and to enumerate them all would be impractical. For this reason, the fundamental concepts behind the GA were introduced in this section. What was not discussed however, was the formulation of the solution space. This involves the design of the solution candidate (bits, reals, integers, etc), as well as how a candidate should be perturbed in solution space (mutation), and how it should be recombined with other candidates (crossover). For self-containment purposes, the most basic implementations of these will be presented. In the simple case where candidates are composed of bits, mutation perturbs a candidate by flipping a randomly selected bit in the array. Crossover recombines two candidates by choosing a random crossover site, and then exchanging bits after this site with another candidate.

3.2.2 Particle Swarm Optimisation

The PSO was invented by Kennedy and Eberhart in 1995 [142] and improved upon by Shi and Eberhart in 1998 [262]. It was inspired by flocking and schooling behaviour in birds and fish. A population of n-dimensional particles are maintained, each of which represents a unique position in solution space, which for the PSO is traditionally n-dimensional Cartesian coordinate space.

The version of the PSO introduced here is that of Shi and Eberhart who introduced an inertial weight to the original PSO [262]. The recurrence relations which describe the single timestep update of the PSO is given in Equations 3.6 and 3.7.

$$\mathbf{v_i} \leftarrow \omega \mathbf{v_i} + \phi_p r_p (\mathbf{p_i} - \mathbf{x_i}) + \phi_g r_g (\mathbf{g} - \mathbf{x_i}) \tag{3.6}$$

$$\mathbf{x_i} \leftarrow \mathbf{x_i} + \mathbf{v_i} \tag{3.7}$$

Where $0 < i <$ population size. The first term is the inertial term, where $0 < \omega < 1$ is the intertia weight. The last two terms involve relative vectors. The vector $\mathbf{p_i}$ refers to the best n-vector found so far by particle i and $\mathbf{g}$ refers to the best solution found across all particles. The variables ϕ_p and ϕ_g are predetermined constants (learning parameters). Choosing these carefully result in the bias of particles towards the global optimum. The values r_g and r_p are simply uniform random deviates in the range $[0, 1)$.

Early variations of the PSO include the *lbest* and *gbest* varieties [285, 224, 214]. The *lbest* (or "local best") refers to a local neighbourhood, where the "best" is the most optimal solution found within a local neighbourhood.

The *gbest* formulation is essentially the same as the *lbest*, except that the local neighbourhood is large enough to contain all particles at all times, so as to ensure communication between them at all times.

More extensively modified algorithms include the Dynamic Multi-Swarm PSO, augmented by the Quasi-Newton method [164], Stretched PSO [213, 214], the Lévy PSO [242] and the simpler Many Optimising Liaisons PSO (MOLPSO) [218].

A particular interest is given to the MOLPSO, as it is more suited to parallelisation, uses less memory, one less random deviate and therefore requires less computation. Although this variation is perhaps more susceptible to local minima than the *lbest* formulation, maximum performance in timestep computation is very important. Some fitness functions (such as entire candidate agent-based models) require excessive computation and frequently also an averaging regime to circumvent the effects of stochasticity. The velocity update equation of the MOLPSO is shown in Equation 3.8. Immediately, one can see that the term involving ϕ_p is missing. The implications of this are discussed further in Section 3.5.

$$\mathbf{v_i} \leftarrow \omega \mathbf{v_i} + \phi_g r_g (\mathbf{g} - \mathbf{x_i}) \tag{3.8}$$

```
Require: $d \mid d \in \mathbb{I} \wedge d \geq 1$ dimensions
Require: $n \mid n \in \mathbb{I} \wedge n > 1$ population size
  let $\mathbb{X}$ be the $d$-dimensional hypercube which satisfies upper and lower bounds on each dimension
  initialise vectors $x_i$ where $i = 1..n$ to random vector in $\mathbb{X}$
  initialise g with a valid random vector in $\mathbb{X}$
  evaluate the fitness of vector g
  while termination criteria not met do
    for $i \leftarrow 0$ to $i_{n-1}$ do
      calculate the fitness of vector $x_i$
    end for
    g $\leftarrow$ current best solution
    for $i \leftarrow 0$ to $i_{n-1}$ do
      $r_g \leftarrow U[0,1)$
      $v_i \leftarrow \omega \mathbf{v_i} + \phi_g r_g (\mathbf{g} - x_i)$
      ensure velocity is within bounds
      $x_i \leftarrow x_i + v_i$
      ensure position vector is within bounds
    end for
  end while
```

ALGORITHM 4: The MOLPSO algorithm.

To more fully illustrate this method, Algorithm 4 contains further description. This algorithm requires almost half as much memory as the usual PSO with both local and global influence, since there is no need to maintain a list of best solutions attained for every particle. Less computation is necessary because particles no longer have to compute a relative vector, and also incur additional memory fetch penalties and multiplications. Finally, r_p is no longer necessary, removing the need to compute a random deviate (a potentially costly operation as will be noted in Section 3.3). More information can be obtained from the work of Pedersen and Chipperfield [218].

3.2.3 Firefly Algorithm

Xin-She Yang invented the FA in 2009 [299]. It is a metaheuristic optimiser algorithm with homogeneous communicating agents and stochastic space exploration, originally for continuous space. The macroscopic patterns of this algorithm are reminiscent of emergent behaviour in agent-based systems, where microscopic rule-based interaction leads to an apparent system-wide pattern. There are also some similarities between the FA and PSO. Yang also states that the PSO can in fact be obtained in a special case of the FA [299].

The biological phenomenon which inspired the FA was mostly the mating habits of fireflies. The flashing light behaviour of fireflies is associated with mating, which, when idealised with a few assumptions, leads to the underlying principle behind the FA. Fireflies are therefore the agents within the system, and encode vectors (or points) in the search space $\mathbb{R}^n$. The assumptions given in the original FA paper by Yang are [299]:

1. All fireflies are attracted to all other fireflies.
2. Attractiveness of a firefly is proportional to the *brightness* of the firefly.
3. Brightness is affected by the objective function.

Roughly speaking, the fitness of a firefly (or candidate solution) is simply the objective function result. When fireflies interact, they observe each others' fitness after it has been degraded based on the distance between the fireflies, and some system-wide hand-calibrated parameters. This light decay is based on the Lambert-Beer law [155] for light decay through a medium with a certain density. For minimisation problems, the attractiveness of a firefly is therefore inversely proportional to the objective function, whereas for maximisation problems, it is simply proportional.

The interaction between two fireflies x_i and x_j is shown in Equation 3.9. This interaction is repeated for every firefly with respect to every other firefly.

$$x_i \leftarrow x_i + \beta_0 e^{-\gamma r_{ij}^2}(x_j - x_i) + \alpha(d) \tag{3.9}$$

The equation makes clear the stochastic and deterministic aspects of the algorithm. The first term is simply an inertial term reminiscent of parts of the modified Particle Swarm Optimiser by Shi and Eberhart [262]. The second term $\beta_0 e^{-\gamma r_{ij}^2}(x_j - x_i)$ is often named the β-step, and the third term $\alpha(d)$ is named the α-step.

The β-step involves an exponential decay of the Cartesian distance between two particles, modified by a constant amplitude scaling parameter β_0, and also a hand-tuned constant coefficient γ. Finally, the α-step traditionally causes a uniform-random perturbation in coordinate space.

Figure 3.2 shows a system of 65536 fireflies searching the parameter space of the Rosenbrock Function in 3 dimensions. Parameters of the optimiser were altered so as to convey particle movements more clearly. The algorithm involves a similar process as the MOLPSO, in that the fitness values of all particles are updated once per timestep. The particles are also propagated by their velocities, which are modified using Equation 3.9.

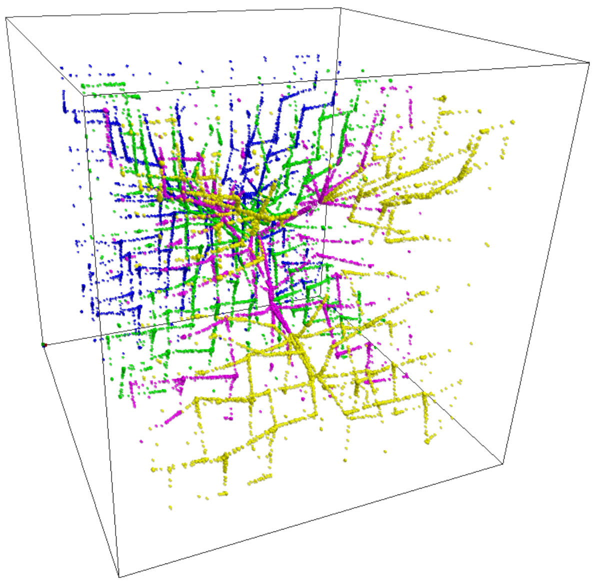


FIGURE 3.2: 65536 fireflies attempting to optimise a 3-parameter generalised Rosenbrock Function. The global minimum is at $(1, 1, 1)$, which is near the centre of the box. The Rosenbrock function is characterised by a low-lying valley, which is easy to find, but the minimum inside this valley is more difficult to find.

3.3 Advanced Space Exploration

Recent effort in improving search algorithms such as the PSO and FA have resulted in more effective exploration behaviour [242, 111] by using Lévy flights.

Lévy flights are random walks where step sizes are determined using a Lévy distribution, giving a combination of long and short trajectories [178]. Brownian motion is the usual random walk in stochastic optimisers, involving some kind of inertia and influenced velocity. The Lévy distribution belongs to the family of stable distributions, along with the Cauchy and Gaussian distributions. Cauchy and Rayleigh flights can be obtained from the Cauchy and Gaussian distributions respectively. The term *Lévy flight* was coined by B.B. Mandelbrot in 1982 [178], alongside *Cauchy flights* for the Cauchy distribution, and *Rayleigh flights* for step sizes drawn from a Normal distribution.

The Lévy α-stable distribution's probability density function is defined as the following inverse Fourier transform [71, 27, 33]:

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(-itx - |ct|^{\alpha}) \tag{3.10}$$

Two parameters modify the shape and width of the distribution. The c parameter scales the width of the distribution, and the α parameter (otherwise known as the exponent) controls the shape and tail. Furthermore, for $\alpha = 1$, it is actually the Cauchy distribution, and for $\alpha = 2$, the distribution becomes Gaussian (no heavy tails). When the distribution is Gaussian and step sizes are sampled from it, this gives rise to Rayleigh flights. With $\alpha = 1.5$ the characteristic heavy tail shape is present, which is the α value used in experiments here unless otherwise noted.

Considering the random walk patterns which Rayleigh, Cauchy and Lévy flights generate gives a qualitative indication of their differences. Figure 3.3 shows sample random walks with 10,000 steps, where step sizes in each dimension are taken from the Cauchy, Gaussian and Lévy distributions. The Brownian random walk and Rayleigh flights appear to have local diffusive characteristics in common, while the Cauchy and Lévy flights have long step lengths in common with longer bouts of small steps causing local diffusion.

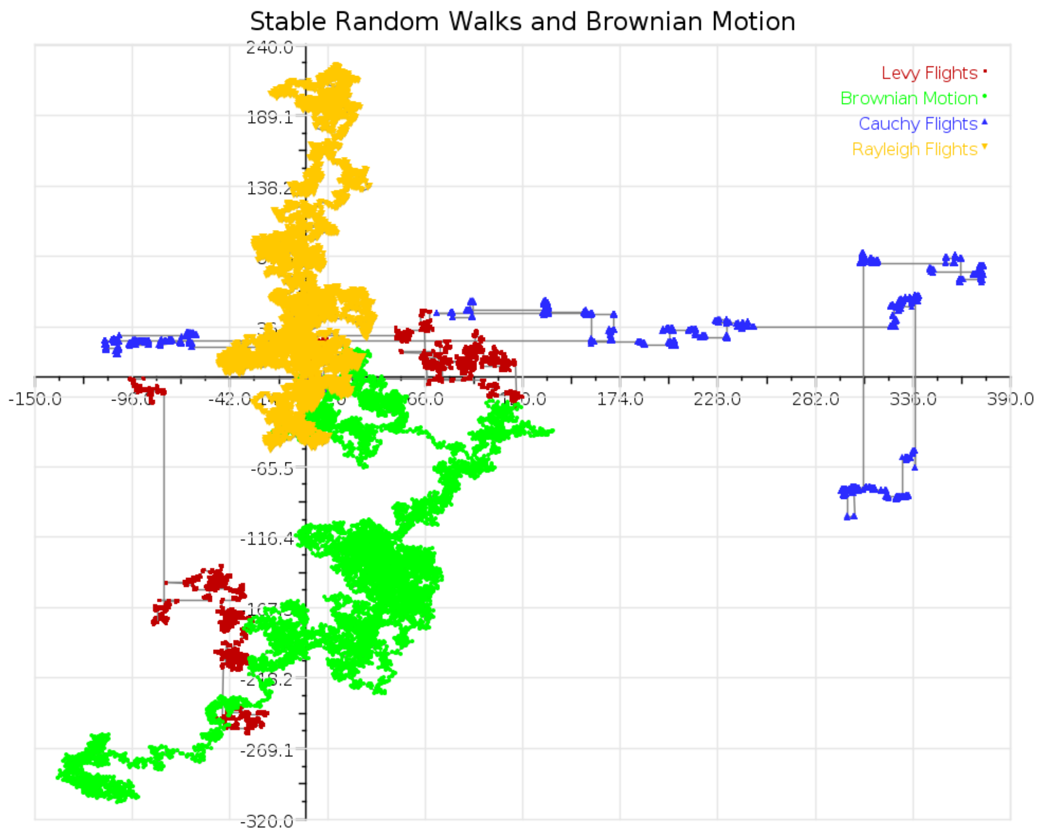


FIGURE 3.3: Random walks generated by Brownian motion and the stable random distributions Cauchy, Gaussian, and Lévy .

The algorithm used for sampling from the Lévy , Cauchy and Gaussian distributions for the purpose of the experiments carried out in Section 3.4 is shown in Algorithm 5 [27, 33][5].

3.3.1 Convergence Results using MOLPSO

The Rayleigh, Cauchy and Lévy flights were implemented in the MOLPSO of Pedersen and Chipperfield [218], in order to test them against conventional Brownian motion. Test functions used and their search space constraints are given in Table 3.1. The boundaries are enforced on all dimensions of the corresponding function.

The parameters of the MOLPSO were as follows:

$$\omega = 0.9, \phi_g = 0.3, c = 2 \tag{3.11}$$

These serve to provide moderate inertia (ω), an attempted balance in exploration and exploitation (ϕ_g) and finally, a Lévy distribution scaling of $c = 2$. A population size of 50 was used. These parameters were chosen experimentally, and may not be the most optimal choice.

[5]For an adapted version of the C code given by Bratley, Fox and Schrage [27], I credit my supervisor Prof. Ken Hawick.

```
procedure levydeviate($c, \alpha$) begin
double u, v
$u = \pi * (U(0, 1] - 0.5)$
$\{$When $\alpha = 1$, the distribution simplifies to Cauchy$\}$
**if** $\alpha$ == 1 **then**
   return $(c \tan u)$
**end if**
$v = 0$

**while** v == 0 **do**
   $v = -\log(U(0, 1])$
**end while**
$\{$When $\alpha = 2$, the distribution defaults to Gaussian$\}$
**if** $\alpha$ == 2 **then**
   return $(2c\sqrt{v}\sin(u))$
**end if**
$\{$The general Lévy case$\}$
return $\frac{c\sin(\alpha u)}{\cos(u)^{1/\alpha}}(\cos(u(1-\alpha))/v)^{(1-\alpha)/\alpha}$
end
```

ALGORITHM 5: Algorithm for generating a Lévy deviate. The algorithm essentially involves a transform applied to two uniform random deviates, effectively consuming two deviates to produce one stable random deviate, using a skewness parameter and an exponent [33].

Two methods of implementation are possible. The first is to use uniform random deviates to choose a direction, and using a Lévy , Rayleigh, Cauchy or uniform (for Brownian motion) deviate for a step magnitude. The other is to simply use a Lévy , Rayleigh, Cauchy or uniform (for Brownian motion) deviate in every dimension. Both methods were implemented, and the results were recorded independently.

	Rayleigh	**Cauchy**	**Lévy**	**Brownian**	**Actual Minimum**
Schwefel 8D	1950 ± 200	1850 ± 200	1950 ± 200	1950 ± 210	0.0
	1950 ± 200	1950 ± 200	1950 ± 200		
Ackley 32D	3.7 ± 0.7	3.9 ± 0.7	3.7 ± 0.7	4.3 ± 0.7	0.0
	3.7 ± 0.7	4 ± 0.7	3.9 ± 0.8		
De Jong 64D	0.05 ± 0.04	0.024 ± 0.004	0 ± 0	3 ± 1.4	0.0
	0.026 ± 0.003	0.104 ± 0.007	0.074 ± 0.005		
Rastrigin 8D	4.2 ± 1.9	4.7 ± 1.9	4.3 ± 1.6	3.9 ± 1.6	0.0
	4.1 ± 2	4 ± 1.8	4.2 ± 2.2		
Rosenbrock 4D	0.5 ± 1.3	0.2 ± 0.9	0.1 ± 0.7	0.5 ± 1.2	0.0
	0.3 ± 1	0.4 ± 1.2	0.3 ± 1		
Griewangk 3D	4.5 ± 2.8	5 ± 3.4	5.4 ± 3.3	5 ± 3.7	0.0
	5.3 ± 3.6	4.9 ± 3.4	5.7 ± 4		
Michalewicz 5D	-4.5 ± 0.2	-4.5 ± 0.2	-4.6 ± 0.2	-4.5 ± 0.3	-4.687
	-4.6 ± 0.1	-4.5 ± 0.2	-4.5 ± 0.2		

TABLE 3.2: Convergence data measured for the MOLPSO with Rayleigh, Cauchy and Lévy flights, as well as Brownian motion. Results were averaged over 300 separate runs, for 3000 timesteps each. Results are in the form $\mu \pm \sigma$, where σ is the standard deviation. Low values denote faster solution. Each combination is given with two results, the first line corresponds to making use of an alpha-step with a uniform-random direction and a magnitude given by either a Rayleigh, Cauchy or Lévy flight. The second line indicates the results obtained from using a Rayleigh, Cauchy or Lévy deviate in every dimension.

Table 3.2 contains the convergence data collected. Each function has a suffix of the form xD, where x is the number of dimensions. Data is presented in the form $\mu \pm \sigma$, where μ is the mean fitness attained at termination, and σ is the standard deviation across the 300 separate executions. Termination criteria is simply 3000 timesteps. Results show that the Lévy flight method gives at least equal or better results than the standard approach as achieved by Brownian motion for space exploration. Results of these flights appear to be similar to those of Richer and Blackwell [242]; albeit with some algorithmic differences. The other random flights do not clearly distinguish themselves from each other. Furthermore, it appears that using a Rayleigh, Cauchy or Lévy flight in every dimension is not as effective as using a single one for a magnitude. This is fortunate, since these deviates are computationally expensive to obtain.

3.3.2 Discussion

The hypothesis that Lévy flights could assist in space exploration was inspired by another natural phenomenon: the flight patterns of insects [300]. However, there is disagreement in the conclusion that Lévy flights cause observed phenomena in ecology [54]. The statistical difference in performance as shown in Table 3.2, warrants some form of explanation, however. The basic assumption is that the foraging patterns of insects essentially allow the algorithm to escape local minima more easily [242].

Very similar concepts in use elsewhere such as Simulated Annealing [145] suggests that adapting step sizes by a monotonically decreasing temperature variable may both explain the somewhat inconclusive results in Table 3.2, as well as create a basis for a greatly improved algorithm. This would be an interesting topic for future work.

Computing correct deviates from a Lévy distribution is relatively computationally expensive. The algorithm of Chambers et al. [33] requires several trigonometric functions, and although simplifications are possible for the special cases where $\alpha = 1$ and $\alpha = 2$, it still typically consumes two uniform random deviates to compute one Lévy deviate. A common method in the literature is to sample from a power law distribution with a heavy tail [300, 111] which is considerably cheaper, but less accurate. Another method uses two normal random deviates to compute a Lévy deviate using a number of linear and nonlinear transforms [242]. It is beyond the scope of this work to compare these methods, however. It is clear that the effective use of Lévy flights demands some careful consideration regarding performance.

3.4 Parallel Implementation

($\mathcal{I}$, pp. 6 & 24)

Population-based EAs are particularly well suited to being parallelised in the particle/candidate evaluation stage. Due to their effectiveness in optimising continuous objective functions, Particle Swarm Optimisers have been some of the first EAs to benefit from parallelism [286, 304, 256, 195] especially with the use of Graphical Processing Units (GPUs). In essence, parallelism in continuous global optimisation algorithms is motivated by their inherently parallel operation, and also the existence of objective functions which are excessively compute intensive. In the context of optimising the parameters of an agent-based model, one is also faced with averaging objective function measurements in order to overcome potential stochasticity in the model, not to mention the potential of computing hundreds of timesteps to reach a suitable system state to obtain a measurement of some form.

Parallelisation of the PSO or the FA is met with some interesting problems. Firstly one must decide on an parallel computing platform. The Compute Unified Device Architecture (CUDA) architecture of NVIDIA is given special interest here due to its inexpensive wealth of theoretical computing power. Making use of the full potential of these devices is also an interesting problem, and deserves thorough investigation. OpenCL [270] and MPI [206] are both software architectures able to make use of graphics hardware (among other architectures). CUDA is the preferred architecture in this work due to its excellent maturity and documentation support, as well as allowing developers access to the internals of the hardware such as all memory banks, universal addressing, and features such as pinned host memory and reentrant kernels [202]. Hardware heterogeneity causes less than peak performance in some parallel platforms [270].

Unless otherwise noted, testing of these algorithms were done using an Intel Core i7 server (3.4GHz), configured with two NVIDIA GTX 590 graphics cards.

3.4.1 MOL Particle Swarm Optimiser

As mentioned earlier, the PSO was the centre of much research in high performance GPU implementations early on [256, 286, 195, 304, 13, 112]. Some of these are GPU-based methods, and some are cluster computer methods. Clearly however, there are different methods of parallelising the PSO. The earliest parallel PSO was that of Schutte et al. [256] in 2003, which involved the division of work among nodes on a Beowulf cluster using MPI (Message Passing Interface) [206]. Venter and Sobieszczanski-Sobieski also made use of MPI in 2005, and introduced another parallel PSO which makes use of a deliberate lack of synchronisation to improve parallel efficiency [286]. The simplicity of the PSO algorithm and its inherent parallel nature proved to be very attractive.

In latter years where GPUs have gained increasing interest, other authors have also proposed GPU-parallel PSO algorithms. These include the work of Bastos-Filho et al. who proposed a GPU-based PSO with a variety of communication topologies, both synchronous and asynchronous [13]. Another named the SyncPSO was developed by Mussi et al. which embraces the limitations of blocks in terms of resources and synchronisation [195].

```
allocate and initialise enough space for $n$ solution vectors with $d$ dimensions on device

allocate space for $dn$ random deviates
while termination criteria not met do
    call CURAND to fill the random number array with uniform deviates in the range [0,1)
    copy vectors to device, including g
    BEGIN CUDA KERNEL
    with $dn$ CUDA Threads ($i = 0..dn - 1$);
    // Safety check for unconventional thread grid configurations.
    if $i < dn$ then
        if velocity thread then
            // Thread grid is set up for separate threads to compute velocities and positions for simplicity.
            $v_i \leftarrow \omega \mathbf{v_i} + \phi_g r_{gi}(\mathbf{g} - x_i)$
        end if
        __syncthreads()
        if position thread then
            ensure velocity is within bounds
            $x_i \leftarrow x_i + v_i$
            ensure position vector is within bounds
            if $i\%(2d + 1) == 2d$ then
                calculate the fitness of vector $x$ which $i$ belongs to
            end if
        end if
    end if
    END CUDA KERNEL
    copy new vectors to host
    obtain the best solution and assign to g
    visualise the result
end while
```

ALGORITHM 6: The parallel implementation of the MOL PSO.

As discussed in section 3.2.2, the MOLPSO provides a few benefits which make parallelisation easier. Algorithm 6 shows a GPU-parallel version of the MOLPSO, where each particle has one thread dedicated to one of its components. Much of this algorithm is dictated by implementation details of the typical GPU-enabled algorithm. Firstly, __syncthreads() is necessary in order to avoid race conditions between threads, which will ensure the new velocity is computed and ready for use. It is worth noting that a simple modification using a previously calculated velocity can avoid this synchronisation. Other race conditions are eliminated by separating read-only solution vectors and write-only solution vectors. Every time step ensures that the device contains the current best candidate in global memory where all thread blocks can access it (**g**). Each dimension of each particle is assigned a thread, maximising the fine-grained parallelisation of the algorithm. Once the velocity update and position update equations are computed, all threads apart from one per particle are disabled. This is done in order to compute the fitness of the particle without race conditions. It is noteworthy that memory copies to and from the device is a very expensive operation. Mussi et al., for example, have opted instead for an algorithm which operates entirely on the GPU without synchronisation with the host other than when it has reached a maximum number of generations [195].

The reason for having identical `if` statements in the algorithm is to ensure that all threads reach the barrier synchronisation. This is a necessary condition to avoid undefined behaviour in CUDA [202]. Finally, the vectors are copied to the device such that position and velocity vectors are adjacent, in order to improve memory locality and retrieval speed.

Lévy flights are also used in Algorithm 6. Enough random deviates are generated by CURAND (which is included as part of CUDA), in order to compute a Lévy deviate for every dimension, for every particle on the device.

In testing this algorithm, 2048 particles were used. This would be a considerably difficult task for a single-threaded implementation. For the Lévy deviates, the parameters used were $c = 2$ and $\alpha = 1.5$ to give a balance between Cauchy and Rayleigh flights. ϕ_g was set to a constant value of 0.01 and particle velocities were constrained to between -0.04 and 0.04 in every dimension.

	Schwefel 4D	Rastrigin 32D	Ackley 64D	de Jong 256D	Rosenbrock 32D
Success % (Original)	1%	0%	0%	0%	0%
(Lévy Flights)	100%	100%	52%	100%	0%
Best Solution	17 ± 39	110 ± 80	1.4 ± 1	1.5 ± 2	35 ± 4
	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	30 ± 1
Timestep time (μsec)	140 ± 2.5	1240 ± 6	2221 ± 0.9	3700 ± 8	632 ± 0.5
	173 ± 0.7	1380 ± 7	2565 ± 1.8	4723 ± 2.6	910 ± 30
Total Timesteps	4000 ± 400	4000 ± 0	4000 ± 0	4000 ± 0	4000 ± 0
	2000 ± 430	213 ± 21	2000 ± 1600	1300 ± 390	4000 ± 0
Total Time (msec)	560 ± 27	4950 ± 20	8890 ± 4	14800 ± 35	2530 ± 2.35
	330 ± 70	300 ± 30	6000 ± 4300	6000 ± 1800	3700 ± 130

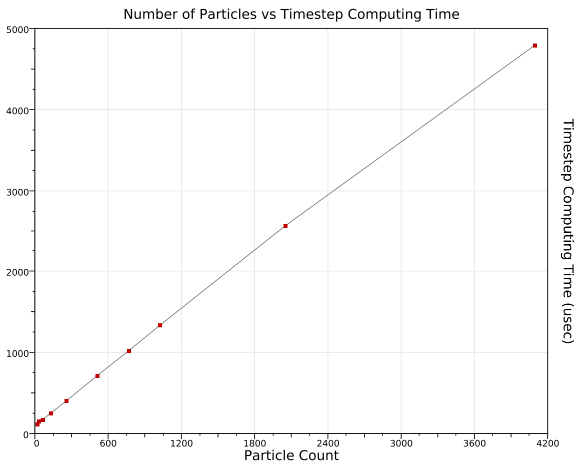
TABLE 3.3: Results with the MOLPSO (in parallel) with 2048 particles including Lévy-flights (3000 frames each, averaged over 100 runs each) accompanied by standard deviations for optimising various test functions. For the measurements collected, the first line pertains to the typical Brownian random space exploration method, and the second is the Lévy-flight method.

The results from comparing the parallel MOLPSO with regular uniform-random deviates against the version with Lévy-flights are shown in Table 3.3. For each measurement collected, the first line pertains to the original Brownian motion space exploration method, and the second pertains to the Lévy-flight method. The most difficult function to optimise was the Rosenbrock function in 32 dimensions. It is hypothesised that variable dependence in the function and a generally neutral fitness landscape contribute to this.

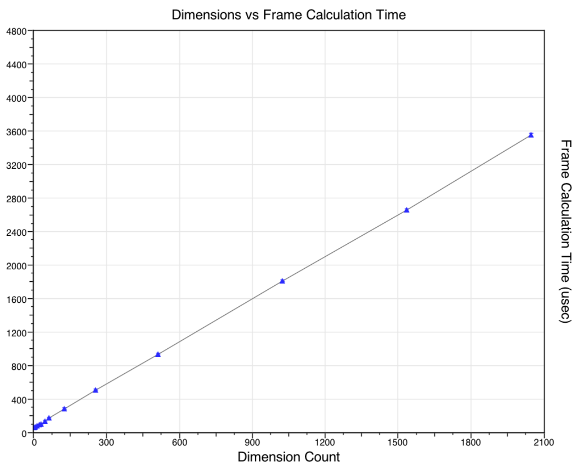
The Lévy flights method seems to have an advantage over the original method. As noted earlier, the computational expense involved in computing Lévy deviates is high over the cost of uniform random deviates. However, the results in Table 3.3 do not indicate that this increased cost is excessive. Given faster convergence, the end result essentially means that the global minimum is obtained in fewer time steps, which is perhaps made more clear by the total time taken. Less total time is necessary in order to obtain the solution than with lower quality space exploration.

In all cases, the original space exploration method terminated at the maximum number of frames (4000), and did not succeed in consistently minimising any of the functions for the given parameters. Success was defined as reaching within 0.0001 of the function minimum.

In a brief additional comparison, the GPU and CPU algorithms were used for De Jong's sphere function in 256 dimensions, with 2048 particles. The CPU algorithm required roughly 65msec for one timestep, and the GPU algorithm required 3.7msec for one timestep. This is roughly an 18X speedup. It is important to take into account that the CPU algorithm can perform much faster with fewer particles, and compute many more timesteps for the same amount of time. However, it is more likely to succumb to typical difficulties such as deceptiveness in fitness functions.



(a) A plot of the particle population size against the frame calculation time for varying numbers of particles with 64 dimensions each, in μ-seconds. The function used here is the Ackley function.



(b) A plot of the number of dimensions against the time taken to compute one timestep with 64 particles (averaged across 100 timesteps in a run) in a range of dimensions on the Ackley function (averaged over 100 separate runs).

FIGURE 3.4: Parallel MOLPSO: Particle population size scaling and dimension count scaling characteristics.

Figures 3.4(a) and 3.4(b) show some scaling performance results. The objective in these tests were to obtain a measure of population-scaling characteristics and also how the algorithm responds to higher dimensions. It appears that there is a fairly linear scaling with system sizes up to 4096. The fact that the MOLPSO is not of complexity $\mathcal{O}(n^2)$ but more of the order $\mathcal{O}(n \log n)$ due to a lack of interaction except with the global best solution means that the algorithm will scale very well. In addition, having one thread assigned to each dimension of every particle further reduces the scaling coefficient in Figure 3.4(b).

3.4.2 Firefly Algorithm

($\mathcal{I}$, p. 47)

Parallelising the FA is met with considerable difficulty [118]. System scaling remains a substantial problem even after attempting to rectify the excessive computation required by unrestricted communication between agents. It is therefore prudent to examine how to reformulate the optimiser such that it is more suited to parallelisation, while minimising the loss of effectiveness. The modification discussed here involves truncating the neighbourhood topology of each particle (agent) in order to reduce the number of interactions necessary. The basis for this is the exponential decay function which degrades perceived fitness across a distance. Assuming that particles at a great distance have negligible influence on the movement of a particle, then these can be removed without causing significant differences in convergence. This does however introduce a new tradeoff, whereby one must ensure neighbourhoods are not too small (so as to cause stagnation), and not too large (so as to reintroduce all interactions and hence the original $\mathcal{O}(n^2)$ complexity.

For convenience, the Firefly update formula is repeated here:

$$x_{i+1} = x_i + \beta e^{-\gamma r_{ij}^2}(x_j - x_i) + \alpha(d) \tag{3.12}$$

The modification made is to choose γ as follows:

$$\gamma = \frac{\ln h}{-g^2}$$

where $h \mid h \in \mathbb{R} \land 0 < h < 1$ is the coefficient given to the β-step when a firefly is observed on the boundary of the local neighbourhood at distance g. Experimental testing revealed that a more smooth decay towards the boundary of the neighbourhood had little benefit over a simpler 0 to 1 cutoff at the grid box boundary. To improve compute performance, the zero-one cutoff was used instead of the light decay function.

To compare against a single-threaded implementation, an implementation written by Mancuso was obtained [177]. Four test functions were used to compare these algorithms, the Rosenbrock, Rastrigin and Schwefel functions, as well as Ackley's Path function.

	Rosenbrock	Rastrigin	Schwefel	Ackley's Path
GPU Time (msec)	9488.7	966.5	848.6	949.9
GPU Minimum	0.000045	1.1741	25.475	3.061
CPU Time (msec)	368460	367329	369935	368384
CPU Minimum	0.000071	1.445	73.267	1.1382

TABLE 3.4: CPU vs GPU Parallel Firefly algorithms in optimising a set of 3-parameter test functions.

Figure 3.5 shows a screenshot of the GPU-based FA, deliberately slowed by small step sizes in order to accentuate the movement characteristics of the fireflies. Thanks to the spatial partitioning techniques developed in Section 2.2.2 in the previous chapter, this algorithm runs very quickly for such a colossal population size.

Table 3.4 contains some performance data for the comparison between the original CPU FA and the GPU Parallel Firefly algorithm presented here. The data collected was averaged over 100 independent runs. Both CPU and GPU algorithms maintained populations of 4096 particles and 600 time steps were executed for one run on each. The global minima for each test function in three dimensions were as follows:

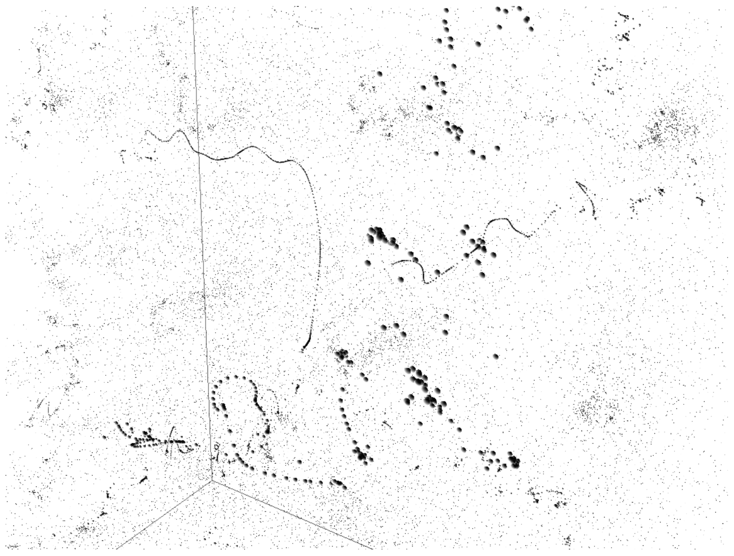


FIGURE 3.5: A population of 262,244 fireflies optimising a 3-parameter Rosenbrock function. Step sizes were deliberately smaller to accentuate movement through space and discovery of better solutions.

1. Rosenbrock function:
   for $f(x, y, z)$, $f(1, 1, 1) = 0$.
2. Ackley's Path function:
   for $f(x, y, z)$, $f(0, 0, 0) = 0$.
3. Rastrigin function:
   for $f(x, y, z)$, $f(0, 0, 0) = 0$.
4. Schwefel function:
   for (x, y, z), $f(420.9, 420.9, 420.9) = 0$[6].

Both the CPU and GPU implementations randomly distribute particules in the allowable ranges of each function shown in Table 3.1. The boundary checks in the CPU Firefly algorithm were removed to more closely resemble the implementation of the GPU algorithm[7]. However, the CPU algorithm was implemented using double precision, whereas the GPU algorithm uses single precision. With the large margin of performance difference indicated by the results, this difference in implementation is less likely the cause. Such a considerable speedup is certainly worth the effort considering the size of the population being used. Saturating the search space with particles certainly assists the algorithms in finding the global minimum, especially in the Schwefel function, where the traditional bounds are $-500 < x_i < 500$.

Overall, both the best compute time and accuracy of the solution were achieved by the parallel algorithm, except the Ackley Path function in accuracy. There are a number of reasons for this. Firstly, it may well be that the Ackley Path function is simply more suited to an optimiser with global interaction, such as the original CPU-based FA. By observation, once the parallel algorithm's particles prematurely converge, they are completely out of reach of others in a different grid box, and zero *useful* interactions will take place. In the case of the single-threaded algorithm, all

[6]The global minimum of the Schwefel function is -3351.8632, but to yield a minimum of zero, this was renormalised so that the minimum is at 0.

[7]Less boundary checks require less computation, and in this case, it was not necessary to force particles to remain within the feasible region.

particles are always in contact with each other, and when premature convergence occurs, it may still be possible for particles to escape and move closer to the global minimum.

The speed-up obtained of the GPU over the CPU is 39 times for 4096 fireflies, but would be much higher for larger numbers of fireflies. Larger bounding boxes such as $-500 < x_i < 500$ would suit even larger numbers of fireflies. In these experiments, it is difficult to form conclusive comparisons (as in metaheuristics in general), but in testing, it was not possible to use the CPU FA for optimising the Schwefel, Rastrigin, and Ackley functions in reasonable time. The GPU FA discussed could optimise all three within one second.

3.4.3 Discussion

It is possible to achieve good performance with far less fireflies. In testing these algorithms, what seemed important is whether local minima were present in the test functions. Functions lacking many local optima tended to be easy for the CPU implementation to optimise with very few fireflies and in very short time. Using 16 fireflies, the single-threaded algorithm could achieve an error of less than 0.00005 in approximately 25msec (on a 10-run average) on the Rosenbrock function. By comparison, the parallel FA discussed here takes approximately 338msec (10-run average) to obtain an error less than 0.00005. The CPU FA has a clear advantage here. However, using 16 fireflies in a complex (local minima containing) function such as the Schwefel or Rastrigin functions, the CPU algorithm will either take an inordinate amount of time (days or weeks), or fail to achieve the global minimum due to premature convergence to local minima.

The basic principle at work in the parallel FA is that of divide and conquer. By saturating the parameter space between the allowable constraints with 2048 or more fireflies reduces the maximum distance between the true global minimum and the nearest firefly. While this still does not provide any guarantee, it greatly increases the chance of success. It is important to note that this is valid for constrained optimisation, but different strategies would be necessary for unconstrained optimisation, where no bounding box is supplied.

Little discussion has taken place on the choice of constant parameters in these algorithms. For the MOLPSO and the FA, at least 3 parameters needed to be calibrated by hand. The choice of these were made empirically by observing the effects on a visualisation of the optimiser. Thoughtful consideration is necessary when choosing these parameters, as in some cases, the wrong choice can lead to consistently suboptimal solutions. The next section deals with the intriguing problem of reinterpreting this hand-calibration effort as another optimisation problem.

3.5 Calibrating Metaheuristic Optimisers

Meta-optimisation has been known by many names including Meta-Evolution and Automated Parameter Calibration [217][8]. It involves the use of an optimiser to calibrate the parameters of another optimiser. Several interesting properties surface from this practice, and several considerable problems which severely hinder the successful use of this method.

An optimiser can behave radically different depending on the parameters chosen for it. It can make the difference between total failure and a consistently low error rate. Consider the PSO update equation, reproduced here for convenience:

$$\mathbf{v_i} \leftarrow \omega \mathbf{v_i} + \phi_p r_p(\mathbf{p_i} - \mathbf{x_i}) + \phi_g r_g(\mathbf{g} - \mathbf{x_i})$$

[8]Instead of providing a thorough introduction to meta-optimisation the reader is referred to Chapter 3 of Pedersen's PhD dissertation on *Tuning & Simplifying Heuristical Optimization* [217].

Here, ω, ϕ_p and ϕ_g are all user-defined parameters. By considering the extreme cases, one can gain an appreciation of precisely how these parameters affect the outcome. For instance, if $\phi_g = 1$, this could cause the particle to gain a huge velocity increase and cause huge overshoot, instead of gently pursuing the global best, which will considerably increase convergence time. If $\phi_g = 0$, no notice would be given to the best solution found globally, and would cause a complete reliance on local information to ensure proper convergence. Similar effects would result from altering ϕ_p in this manner. In summary, there may be sets of parameters which are similar in effectiveness, as well as sets which differ dramatically [180].

Given also that different combinations of these parameters can be effective for different problems, it is reasonable to view this as an optimisation problem as well: *meta*-optimisation [217]. This practice has been researched in some depth already, but remains somewhat constrained by excessive compute times. Previous works include those of Meissner et al. who have considered using an overlaid optimiser [180] to improve the PSO, Pedersen's meta-optimiser for the Differential Evolution algorithm [216], and Kantschik et al. who used the concept of meta-evolution in order to evolve recombination operators themselves, essentially evolving key components of the optimisation algorithm [139]. Clearly the results of this practice can be useful.

Similar to the problem of calibrating an agent-based model, the compute time problem in meta-optimisation stems from the fact that optimising a complex "black-box" function takes a certain amount of time. Stochasticity also demands several iterations in order to obtain an average. For this reason optimisers are normally best suited to trajectory-based methods such as Local Unimodal Sampling [217], or Simulated Annealing [145], as there would be only one candidate that would be iteratively improved instead of an entire population; as would be the case in the PSO.

Preference is given to population-based methods, due to the benefits they possess from communication between agents. Even if a population of interactionless trajectory-based optimisers were used, the computing power of parallel devices such as GPUs offer an intruiging improvement to performance. Trajectory-based methods do also deserve further discussion in the context of parallelism, but the focus is given to population-based methods here, for their inherent parallelism and communication strategies. In this section, the MOL PSO is used due to its implementation simplicity and performance characteristics as detailed by the previous sections.

3.5.1 Methodology

In meta-optimisation, the overlaid optimiser is sometimes known as the super-optimiser (SRO), and its population (termed super-particles) consists of particles in $\mathbb{R}^n$. In this case, the SRO is implemented as a MOLPSO, and its parameters (as shown in Equation 3.8) are ω and ϕ_g. The effect of the c parameter on the Lévy distribution is scaling, which will affect the step sizes taken dramatically, therefore, it is included as part of the sub-optimiser (SBO) parameters. Therefore, the spatial position of super-particles determine the parameters (ω, ϕ_g, and c) of their respective SBOs. The evaluation phase of this SRO is what sets it apart from ordinary optimisation, in that the candidates are themselves optimisers.

Figure 3.6 shows the conceptual operation of an SRO. Sub-optimisers are not involved in the SRO anywhere except in the evaluation phase. Here, $x, y \in \mathbb{R}$, and when each particle must be evaluated, these values are propagated into a small unique MOL PSO with $\omega = x$ and $\phi_g = y$ hypothetically. In order to return a fitness value for the super-particle at (x, y), the sub-optimiser initialised is trialled by attempting optimisation of the Rosenbrock function, for instance.

For testing the parallel implementation of the meta-optimiser discussed here, the MOLPSO super-optimiser is set up with 16 particles, each with initially random locations in a cube where the principal axes (x,y,z) are

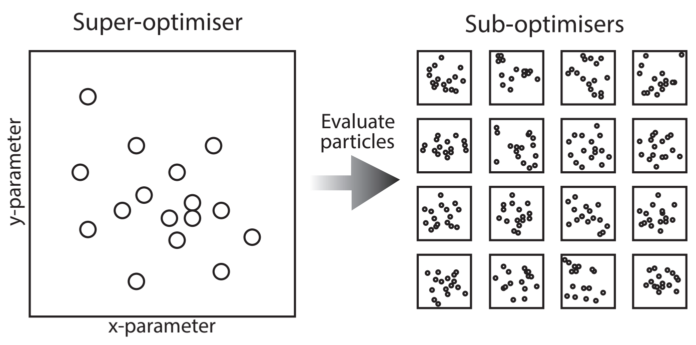


FIGURE 3.6: A pictorial representation of the relationship between the meta-optimiser, and the fitness evaluation of its population.

constrained to -4 and 4. This boundary seems to be appropriate enough to observe a range of effective and ineffective optimisers. The axes x,y and z represent the sub-optimiser parameters ω, ϕ_g and c respectively. The parameter c is directly used to scale the size of Lévy flights (discussed in Section 3.3), given that these have a dramatic effect on the effectiveness of the optimisers.

Let P be the set of super-particles, and let $S(p)|p \in P$ be the fitness of a super-particle at position $p = (x, y, z)$. To evaluate $S(p)$, a CUDA kernel calculates 500 time steps of the sub-optimiser where $\omega = x$, $\phi_g = y$ and $c = z$. The output fitness value is the highest value of the test function obtained, which is averaged across 20 attempts. In summary, to compute one time step of the SRO, $S(p)$ must be computed for all $p \in P$, which involves 500 time steps computed 20 times to obtain an average. It is important to average the results in order to ensure stochasticity does not compromise the accuracy of the fitness obtained [180].

In essence, CUDA is used to evaluate all the candidate optimisers with their respective parameters on GPU hardware. This process of evaluation significantly outweighs any computation done at the super-optimiser level.

The number of particles in the sub-optimisers is set to 20, and velocities are constrained to a maximum magnitude of 1.0, and initialised randomly. The sub-optimisers are evaluated by several test functions, given in the next section. 300 time steps of the super optimiser were computed, and this was repeated 40 times to obtain an average.

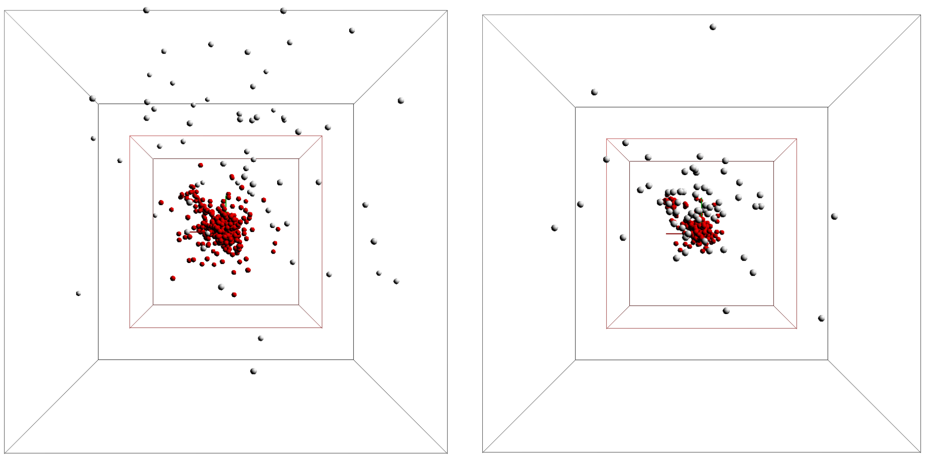
Finally, super-particle velocities are also constrained to less than 1.0, and other hand-calibrated parameters for the SROs are:

$$\omega = 0.9, \;\; \phi_g = 0.03, \;\; c = 4.0 \tag{3.13}$$

3.5.2 Results

A visualisation of an in-progress meta-optimiser is shown in Figure 3.7. Lighter grey particles and the lighter grey bounding box represents the realm of the SRO: the axes of the parameters ω, ϕ and c. As seen here, several of the super-particles are towards the centre of the cube, indicating that smaller values of each parameter are perhaps more suitable. The darker red particles represent the best averaged fitness obtained from the sub-optimisers. The super-particles and the sub-particles are superimposed within the same region.

In the author's experience [124], it was necessary to make small differences in ω and large differences in c in order to optimise the MOLPSO for different test functions. Only one function was used to measure fitness in sub-optimisers at a time, however.



(a) First timestep. (b) Several timesteps later.

FIGURE 3.7: A snapshot of the GPU-based SRO (grey particles) and the end-resulting particle distribution of sub-optimisers (dark-red particles) after several frames of the SRO.

	Rosenbrock 8D	**Schwefel 8D**	**Griewangk 16D**	**Michalewicz 5D**	**Ackley 16D**
Meta-PSO					
Mean Result	4.05	-1941	2.3	-3.69	0.0
Std. Dev.	0.14	32.4	0.13	0.011	0.0

TABLE 3.5: The mean results generated by the MOLPSO meta optimiser for several test functions. Low values denote higher quality solutions. Function boundaries are given in Tab. 3.1.

	Rosenbrock 8D	**Schwefel 8D**	**Griewangk 16D**	**Michalewicz 5D**	**Ackley 16D**
ω	0.2557	3.8690	1.7098	-0.3540	0.6784
ϕ_g	2.1863	3.0114	4.0	0.3332	0.4677
c	0.7154	4.000	3.5909	3.8098	0.3988
$F(\mathbf{x})$	3.7471	1330.7	1.9462	-3.6953	0.0000

TABLE 3.6: Best sub-optimiser parameters generated by the meta-optimiser throughout the 40 separate runs.

Table 3.5 shows convergence results for the test functions used, which were obtained by averaging $S(p)$ across 40 independent runs of 300 time steps of the SRO. This table is discussed below.

Of the test functions used, the Ackley function in 16 dimensions appears to have been the easiest to optimise. The ability of the meta-optimiser to generate parameters (given in Table 3.6) which allow a 20-particle optimiser to consistently optimise the Ackley function is impressive, as it requires a considerable amount of computing power to obtain these parameters.

In previous experiments (see [124] and Section 3.3), acceptable values for ϕ_g were typically in the range $[0.01, 0.1]$. As shown in Table 3.6, ϕ_g and ω are quite large. Larger step sizes were expected for the Schwefel and Griewangk functions due to the large boundaries involved, however. This behaviour is made clearer in Figure 3.8.

The excessive freedom allowed in the bounds of the parameters ω, ϕ_g and c allowed the SRO to seek out unconventional parameters as well. The c value for the Schwefel function is one example. For ease of reference, c is a scaling parameter of the Lévy distribution of step sizes. Larger values corresponds to generally larger steps, which in the case of the Scwhefel function in Table 3.5 is very large, perhaps indicating a need for larger step sizes. It can also, however, be a symptom that the sub-optimiser's population is simply not substantial enough to effectively search the landscape of the vast search space of the Schwefel function.

During testing, the average time taken to compute one timestep of the super-optimiser was approximately 150msec. Larger numbers of particles quickly increased this number, however. The purpose of this short study was to indicate that it is indeed possible to effectively accomplish meta-optimisation using GPUs, rather than to demonstrate the precise speedup over a CPU implementation.

3.5.3 Discussion

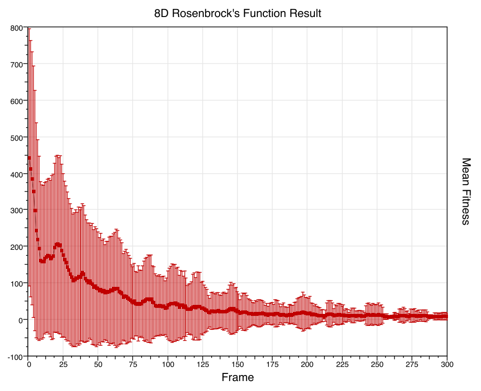
Good results were obtained by using meta-optimisation to find appropriate parameters for optimising certain test functions. However, the process in doing so is still computationally expensive. Due to CUDA block size limitations, it was not possible to extend the sub-optimiser dimension count to more than 16 without a total redesign of the algorithm.

The resulting parameters also appeared in some cases to take advantage of the individual characteristics of the test functions which they were assigned. This is usually an undesirable effect, but could be mitigated somewhat by testing all functions at the same time. Though, this would again increase the computational cost tremendously. This problem is otherwise known as "overfitting".

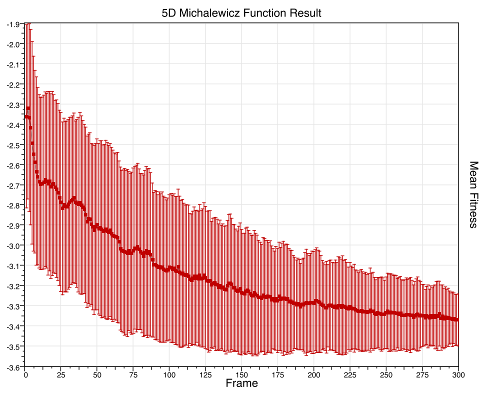
An auxiliary advantage of meta-optimisation is that it facilitates comparison [217] between optimisers.

As shown earlier, certain agent-based models such as flocking models share the inspiration behind optimisers such as the PSO. In much the same way, both the PSO and most agent-based models require some form of calibration. This short study has indicated that meta-optimisation is effectively equivalent to the calibration of an agent-based model. The definition of the fitness function is perhaps more difficult in an agent-based model, but the precise process is demonstrated very well by meta-optimisation of the MOLPSO using another MOLPSO.

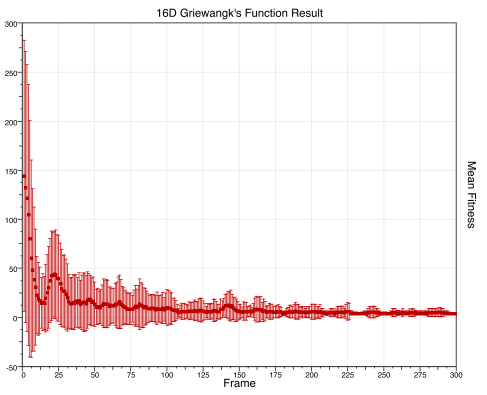
Having discussed optimisers which involve arbitrary numbers of dimensions, and also noting that it is important to have a qualitative sense of how an optimiser operates, it is therefore relevant to discuss visualisation. Higher dimensions above 3D are less trivial to visualise, and require special pre-processing. The next section demonstrates some simple techniques for visualising these.



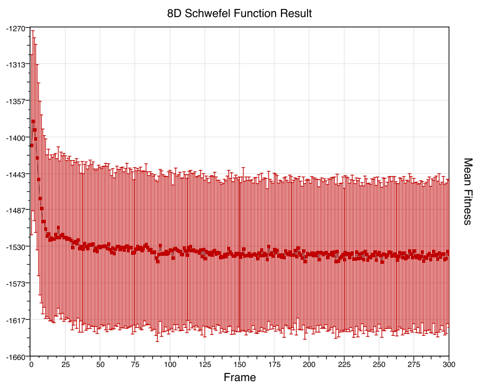
(a) Meta-optimiser on Rosenbrock's function.



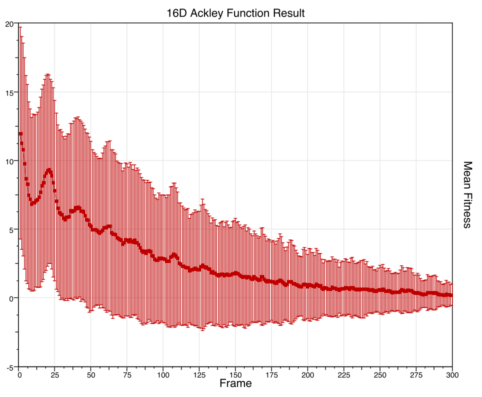
(b) Meta-optimiser on Michalewicz's function.



(c) Meta-optimiser on Griewangk's function.



(d) Meta-optimiser on Schwefel's function.

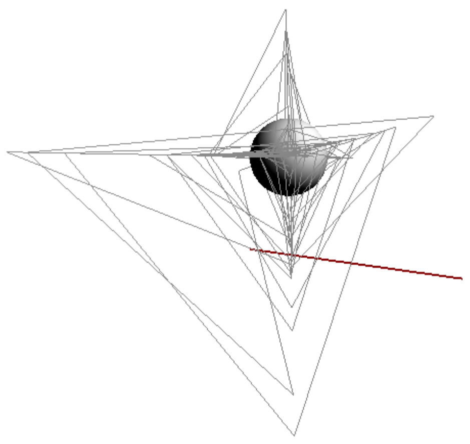


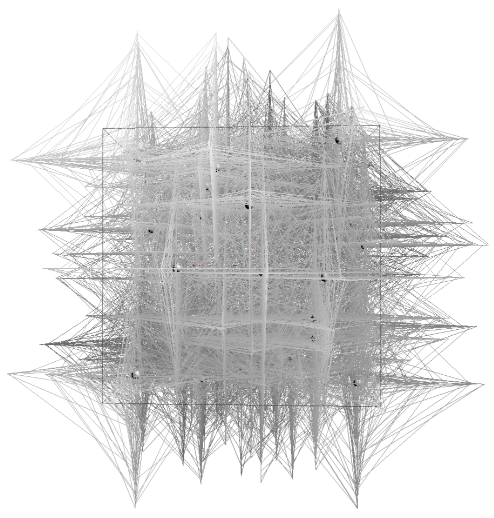
(e) Meta-optimiser on Ackley's Path function.

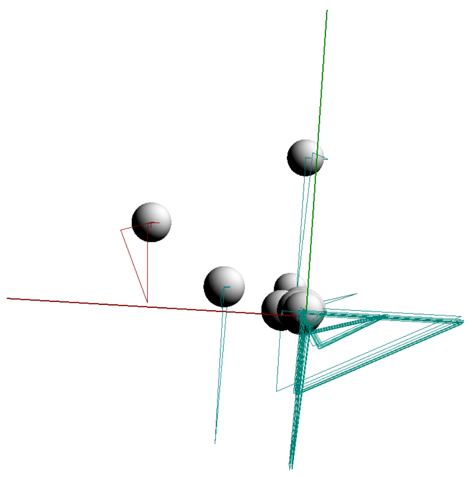
FIGURE 3.8: Plots of the mean fitness obtained from the super-optimiser across all 40 separate runs, including error bars representing the average standard deviations of each point (sub-optimiser) across the separate runs.

3.6 Higher Dimension Visualisation

Visualisation of an optimiser gives a good qualitative understanding of its operation, and can result in valuable information to improve its effectiveness. Low dimensions $d = 1..3$ are straight forward to visualise, but higher dimensions (often the case in global optimisers) $d = 4, 5, 6, ...$ are less trivial. One possible method is to utilise a dimension reduction technique such as Principal Component Analysis (PCA) [69]. A very simple (but perhaps rudimentary) method is given here, which is very easy to implement and does not perform any dimension reduction. Results of this are shown in Figure 3.9.


(a) A single 64-dimension candidate solution.


(b) De Jong Sphere function in 128 dimensions.


(c) Rosenbrock function in 32 dimensions where the optimisation attempt has stagnated.

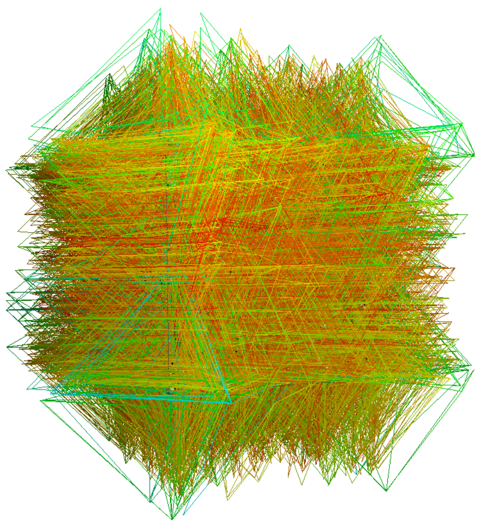

(d) Rosenbrock function in 32 dimensions, with fitnesses coloured using hues of the HSV colour space.

FIGURE 3.9: Visualisation of an optimiser searching for a global minimum for various functions in various dimensions.

An example of a single particle in $\mathbb{R}^{64}$ is shown in Figure 3.9(a). Particles are visualised by first placing a particle sprite in a 3D location denoted by the first three dimensions. This is to ensure later, that when convergence occurs, that it is obvious when more than one local optimum is discovered apart from the global optimum. The next dimension value is used as a magnitude and a line is drawn from the sprite in the x axis with that magnitude. The end of the line becomes the current position, and the next dimension is also used as a magnitude, but from the sprite along the y axis. A line is then drawn from the current point to the next point, and so on, wrapping back around to the x axis after drawing a line to a point along the z axis. Colours are assigned using the HSV colour space assigning a Hue of 0 degrees for the worst solution, and 180 degrees for the best.

The end result is a quick visual display of the particles in $\mathbb{R}^n$. The efficacy of this visualisation is perhaps only appreciated when viewing it in real-time[9]. Collective movement of all these n-dimensional particles show convergence towards a clearly recognisable goal configuration, where the optimum is of a distinctive shape. This is an easily implemented visualisation and provides rudimentary, but effective visual cues from particle-based optimisers.

Figure 3.9(d) shows an example of an optimiser searching for a solution for the 32-dimension Rosenbrock function. Colours in this diagram are used to indicate better solutions. The worst solutions are coloured dark red.

[9]A video of this is available at http://husselmann.com/alwyn/fig3.9.html.

CHAPTER 4 COMBINATORIAL OPTIMISATION

As discussed in Chapter 3, the domain of continuous optimisation is well equipped with a variety of algorithms. Departing from Cartesian real-coordinate space to a combinatorial search space where solutions are combinations of "building blocks" is not a trivial transformation. Nevertheless there have been many algorithms proposed for this task. As an example, there have been some discrete variations to metaheuristic optimisers such as the Firefly Algorithm (FA) [299], which have been applied to interesting and diverse problems such as flow shop scheduling problems [251] and cryptanalytics [208].

The purpose of this chapter is to form the basis for the structural optimisation subsystem in Chapter 6. This chapter begins by introducing Genetic Programming followed by some test problems commonly used in the literature. The use of Genetic Programming and its variants on Graphical Processing Units (GPUs) are also discussed, given performance indications from Chapter 3. The second half of the chapter pertains to the application of Geometric Optimisation, making use of optimisers previously intended for other search spaces. The *Many-optimising-liaisons* Particle Swarm Optimiser and Firefly Algorithm discussed in Chapter 3 are adapted for program search spaces and parallelism. The chapter ends with a short study on program-space visualisation, enabling the visual perception of a population of candidate programs.

The contents of this chapter extend upon work previously published by the author in *Proc. 10th International Conference on Genetic and Evolutionary Methods (GEM'13)*[1], *Proc. Int. Conf. on Artificial Intelligence (ICAI'13)*[2], *Cuckoo Search and Firefly Algorithm, Springer*[3], and also *Proc. Int. Conf. on Information and Knowledge Engineering (IKE'13)*[4].

4.1 Introduction

FACILITATING THE DISCUSSION ON GEOMETRIC-TYPE OPTIMISATION IS MADE MORE COHERENT by first discussing conventional combinatorial optimisation [197]. One might think of the traditional combinatorial optimisation problem as the Traveling Salesman Problem (TSP) [132]. In this chapter, the formulation of

[1]A. V. Husselmann and K. A. Hawick. Genetic programming using the Karva gene expression language on graphical processing units. Technical Report CSTN-171, Computer Science, Massey University, Auckland, New Zealand, July 2013

[2]A. V. Husselmann and K. A. Hawick. Geometric optimisation using karva for graphical processing units. Technical Report CSTN-191, Computer Science, Massey University, Auckland, New Zealand, February 2013

[3]A. V. Husselmann and K. A. Hawick. Geometric firefly algorithms on graphical processing units. In *Cuckoo Search and Firefly Algorithm*, pages 245–269. Springer, 2014

[4]A. V. Husselmann and K. A. Hawick. Visualisation of combinatorial program space and related metrics. Technical Report CSTN-190, Computer Science, Massey University, Auckland, New Zealand, 2013

the search space is defined as the set of all *expression trees* which can be constructed from a set of symbols which result in a tree representing program flow. The objective function is defined by the problem domain and allows easy evaluation of such an expression tree. The goal is to minimise or maximise the objective function, as in continuous global optimisation (see Chapter 3).

Genetic Programming (GP) [150, 151, 149] has become the term used for most population-based search algorithms which operate on the space of interpretable, or directly executable programs or decision trees. These programs are typically constituted by syntax from a tailored and very simplified Domain-specific Language (DSL). The set of GP algorithms are divided by their method of representation, such as tree-based and linear varieties. The former represents candidate solutions as trees which are interpreted at run-time, while the latter represents programs as instructions which are executed (or interpreted) by the computer in sequence.

GP makes use of evolutionary phenomena, where candidates propagate depending on natural selection, genetic crossover and mutation [225]. GP was the ground breaking work of John Koza in 1992 [149], predating the Particle Swarm Optimiser by only a few years [142]. Before this, combinatorial optimisation received much research interest for many years. Classic problems include the Traveling Salesman Problem [132], the Prisoner's Dilemma [6] and the Knapsack problem. Problems such as these are just as applicable in reality as continuous optimisation problems. Research advancements on classic problems such as these are potentially beneficial for many additional problems as well. So far, GP and its variants have been used for intrusion detection [40], soccer robotics [170, 167], as well as land use change modelling [179], multi-agent learning [171], algorithm discovery [283], and image enhancement [223]. It also has applications in cooperative multi-agent systems [210], and classification tasks in data mining [305].

In the past some impressive performance results were obtained in the realm of combinatorial optimisation with regard to high performance [31, 52, 118, 157]. Cano et al. reported an impressive 834X speedup against single-threading, and a 212X speedup against a 4-threaded implementation of Ant Programming [31]. With recent and ongoing improvements in parallel hardware, there certainly is scope for additional research in finding other high performance evolutionary algorithms. It appears that as recently as 2013, combinatorial optimisation was somewhat under studied [255]. Schulz et al. advocate for focus on efficiency and more complex optimisation algorithms in comparison to the simpler GPU algorithms already published [255].

In the search for improvements in this field, many modifications to GP have been proposed since 1992. Some of these include linear representations [25], Cartesian GP [184] and Strongly Formed GP [32] to name a few. There are also other extensively modified optimisers such as Ferreira's Gene Expression Programming (GEP) [64]. Many modifications are intended to alleviate common problems in GP which adversely affect convergence rates. O'Neill et al. give some open issues in GP as follows [205]:

1. Candidate representation
2. Judging problem difficulty (fitness landscape)
3. Dynamic problems
4. Determining how much influence from biology to accept
5. Continuous (dynamic) evolution
6. Generalisation
7. Benchmarking and comparison

8. Scalability and modularity
9. Handling algorithmic and structural complexity

Another important problem is that of code size [266]. “Code bloat” occurs when candidate solutions increase in size due to the sustained introduction of new genetic material by mutation. This is often due to the representation of candidates and the specific formulation of the mutation operator, which is discussed later.

Like population-based continuous optimisers, GP also maintains a population of candidates. Experiments in this chapter require candidate solutions to be expression trees, denoting program flow. There have been many other representations, particularly linear ones [25]. Algorithms directly inspired from Holland’s Genetic Algorithm such as Grammatical Evolution (GE) generate directly compilable code (depending on grammar) from Backus-Naur form (BNF) grammars [249]. GE represents candidates as strings of integers, and the mapping of these to code is called the *genotype-phenotype* mapping. The integers indicate which production rules to use from a given BNF grammar.

The GP algorithm successively improves on a population of candidates by performing *crossover* and *selection* operators on them, in a similar fashion to the original Genetic Algorithm [107]. The old population (or previous “generation”) is replaced, and the new population is then evaluated. Changes in how candidates are represented in GP call for a complete re-engineering of the crossover and mutation operators, however. The implementation of these is often subject to slight variations and many different operators have been proposed [25, 64], but the concept remains the same. The crossover and mutation operators can be thought of as the *exploration* and *exploitation* concepts discussed in Chapter 3, in that mutation causes perturbation of a candidate in the pseudospace of solutions, and the crossover operator exploits good solutions already found. Concrete examples of these are given below in Sections 4.1.1 and 4.1.2.

Crossover

The *crossover operator* in GP deals with recombining two (or more [56]) candidates [197]. In the traditional representation of abstract syntax trees (ASTs), this is done by performing a subtree swap on the two candidate trees on a random point, named the “crossover site”. This attempts to construct two new candidates which can then be used in the subsequent computations. This is very simplistically analogous to the biological process of genetic crossover. There are however other interpretations, some of which are perhaps more true to the natural analogy [64].

In linear representations of candidates, crossover is commonly done by selecting a point in two candidates’ sequences of instructions, and then exchanging the two sequences of instructions about that point with the corresponding sequences of the other candidate. The result is a set of two candidates from the input set of two candidates, which are rearranged.

Selection

The *selection operator* in GP is used to choose two candidates to be provided to the crossover operator as inputs [197]. There are several methods of accomplishing this, the traditional method is fitness-proportional selection (or “Roulette wheel selection”), where the probability of selecting a candidate is proportional to its fitness. Tournament selection is another method, where n “tournaments” are held, where two (or more) candidates are chosen in a uniform random fashion, and then the candidate with the lowest fitness is discarded. This operator ensures that the genetic material of suitable candidates are preserved.

Mutation

The *mutation operator* in GP is the simplest, in which candidates are perturbed in solution space, which is intended to introduce new genetic material [197]. This serves as a method for exploring solution space. Without this operator, the algorithm is only able to cause a drift (with the crossover operator) [63], and it is unlikely the global optimum will ever be discovered.

Briefly, traditional mutation is derived from initialisation methods such as [289]:

1. Grow method: Random successive selection of functions and terminals to build a tree up to a predefined maximum depth.
2. Full method: Random selection of functions and terminals, resulting in a tree where every leaf is at a predefined depth.
3. Ramped half-and-half: a combination of the above methods are used.

Originally, mutation is simply a replacement of a subtree by “regrowing” it using one of these methods. An initial set of candidates are usually constructed using the *Ramped half-and-half* strategy.

Like continuous global optimisation (see Chapter 3), these kinds of optimisation algorithms have their own class of problems which facilitate comparison among the numerous variations already proposed. Two of these used widely in the literature are presented and discussed. Firstly, the classic Symbolic Regression problem, followed by the Santa Fe Ant Trail problem.

4.1.1 Symbolic Regression

A popular testbed for algorithms like GP is Symbolic Regression [4]. This problem serves as a suitable example of the capability of GP. Symbolic Regression is the search for an expression which best matches an unknown function by observations of it; for instance, $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, given a finite set of x and $f(x)$ pairs. An example could be a Sextic polynomial such as:

$$f(x) = x^6 - 2x^4 + x^2 \tag{4.1}$$

Assume this function is unknown and a set of observations is given. The GP algorithm would maintain a population of candidates, where each candidate is initialised to an arbitrary expression, constructed from terminal $\mathbb{T}$ and nonterminal symbols $\mathbb{F}$ (also termed functions). In this case, the sets could be $\mathbb{T} = 0..9$ and $\mathbb{F} = Q, *, +, -, /$ (Q denotes the square-root 1-arity function). Suppose a candidate expression was initialised to $x * x * x * x * x * x + 2x + 3$. By simple observation this is a poor match for Equation 4.1. However, it is important that the algorithm has a quantifiable measure of “fitness”, or suitability, in order to propagate candidates which are more suitable. By feeding a set of inputs through the “black-box” of the real $f(x)$, one can obtain the root mean squared error (RMSE) measure which will indicate a real-valued distance from the candidate’s $f(x)$ expression to the actual $f(x)$ in Equation 4.1. The comparison between candidates in a population is what facilitates “survival of the fittest”, and is hence very important.

It is not necessary to know the actual $f(x)$ during testing; a sample dataset of input/output pairs would be adequate. However, it is convenient to use the actual function while testing the convergence capabilities of algorithms. Typically, a more simple scoring mechanism is used: should the error of the candidate $f(x)$ be less than a certain value ϵ then the test is deemed a “success”, and the score for the candidate is incremented. A perfect solution is a faithful reconstruction of the original $f(x)$, and therefore (ideally) achievement of the maximum score possible.

4.1.2 Santa Fe Ant Trail

The Santa Fe Ant Trail problem is another good problem for evaluating algorithms based on GP [149]. Figure 4.1 shows the initial state of the problem. The objective is to combine specific function and terminal symbols in such a manner that the ant follows the path indicated by the black line, consuming all "food" tokens. Figure 4.2 shows a modified 3D version of the Santa Fe Ant Trail problem with food particles uniform-randomly distributed in a cube in continuous space.

The nonterminal function set is $\mathbb{F} = \{\text{IfFoodAhead (I)}, \text{ProgN2 (P)}\}$, where IfFoodAhead is of arity 2, taking a left operand indicating what to do when food is not directly ahead, and a right operand which is executed when a food token is directly ahead. The ProgN2 nonterminal is of arity 2 and simply executes both its arguments in order. To simplify matters somewhat, the ProgN3 nonterminal is discarded unless otherwise noted. This does however increase complexity somewhat, as multiple ProgN2 functions are necessary to reconstruct a ProgN3 function. The terminal set is $\mathbb{T} = \{\text{Move Forward (M)}, \text{Turn Right (R)}, \text{Turn Left (L)}\}$.

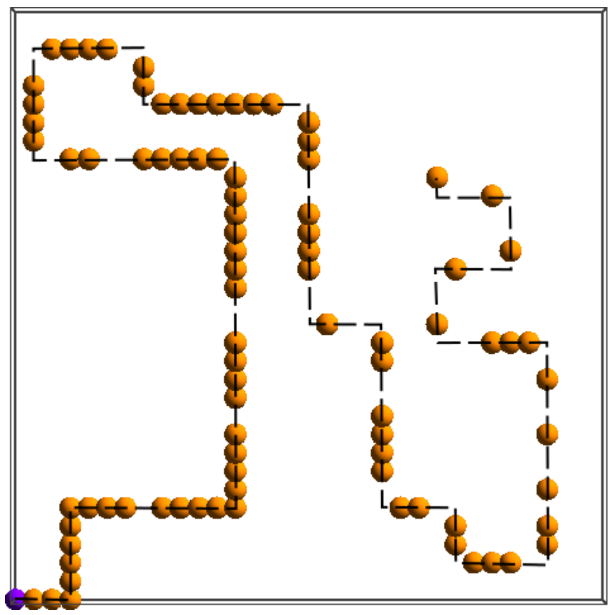


FIGURE 4.1: The initial state of the Santa Fe Ant Trail problem. The ant agent is in the lower left corner of the lattice, and its task is to consume all food tokens while not being able to sense anything but tokens that are immediately adjacent to itself.

This problem may seem trivially simple, but an important limitation is that the ant is not able to sense any food tokens unless it is directly in front of it (also facing it). Consider the very simple attempt at a program to solve this problem in Figure 4.3. The tree is traversed once for each time step. By following this through by hand on the trail shown in Figure 4.1, it quickly becomes obvious that it cannot solve the whole problem, as the ant will stop moving at the first gap in the line.

A program with 100% accuracy generated by John Koza is shown in Figure 4.4. Koza notes that this program contains some redundancy, but does not affect its performance. With regard to the fitness function, Koza also notes that no preference is given to more efficient programs (ie. less redundancy) [149]. Therefore, there is a large number of programs which can solve this problem, some would be functionally equivalent.

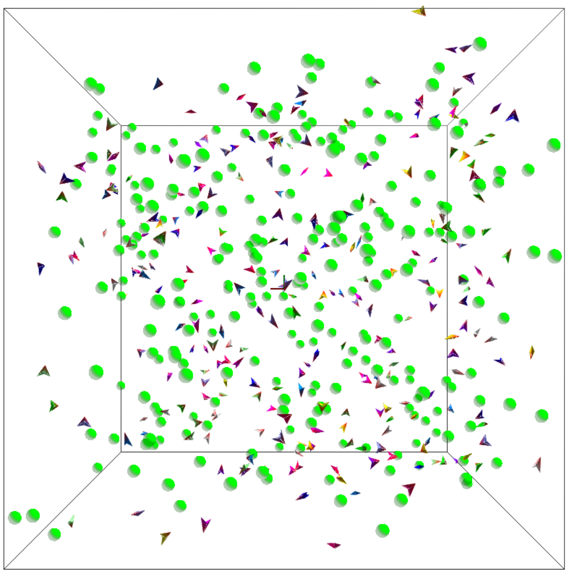


FIGURE 4.2: Initial state of a modified, 3-dimensional Santa Fe Ant Trail problem, with food tokens placed uniform-randomly across a cube in continuous space.

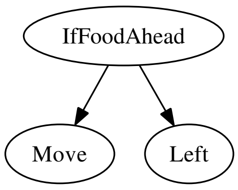


FIGURE 4.3: A very simple attempt to solve the Santa Fe Ant Trail problem. Simulating this by hand will quickly indicate precisely why this problem is a deceptively simple one to solve.

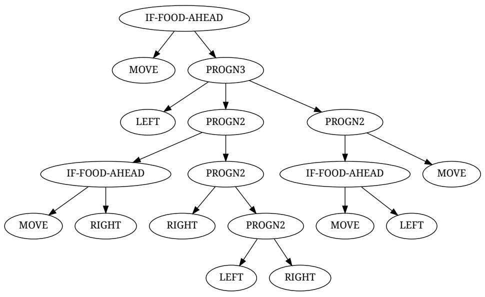


FIGURE 4.4: A perfect solution to the Santa Fe Ant Trail problem generated by John Koza [149].

Having discussed some problems and their symbol sets, it is important to now discuss precisely how a candidate is stored, as this will affect the rest of the implementation.

4.2 Linear Representations and Karva

One of the more prominent problems in using GP is finding an appropriate form or datastructure to encapsulate a candidate solution to a problem. GP candidates are originally (and intuitively) encoded as pointer tree datastructures. This is not conducive to the use of Graphical Processing Units (GPUs) for accelerating them, however. While dynamic memory allocation is now supported by the very successful Compute Unified Device Architecture (CUDA) API, it is not without overhead due to the use of the global memory bank. Race conditions would also plague the construction and traversal of these trees.

There are generally three different candidate representations, they are tree-based, linear and graph representations [25]. The original tree-based representation devised by Koza involved the use of LISP S-expressions [150]. The newer linear [25] representation is more focussed upon a set of instructions, executed one after the other. Graph GP [139] is a generalisation of the tree-based representations, which could perhaps be well illustrated as a flow diagram.

A representation named Karva [63] is discussed here. Strictly speaking, it falls under the category of linear representations. It has some advantages over the original tree-based representations and as such it is used later in this chapter. It is a linear representation of expression tree structures which Ferreira used for developing the Gene Expression Programming algorithm [64] which are stored in memory as strings of symbols. Karva is closely related to Read's linear code [219], which was an early method to linearise expression trees for GP. *Introns* are supported by Karva, which is an interesting feature explained in this section along with how operators such as crossover and mutation are applied. It should be noted that the Gene Expression Programming algorithm features many additional operators including replication, mutation, three transposition operators and three recombination operators. These serve to more closely emulate reality and guarantee effective circulation of genetic information [63].

Another advantage behind the Karva language of GEP is its ability to effortlessly handle any-arity functions, variable length, and introns. Expressions based on Karva in GEP are known as k-expressions, and are constituted by a set of symbols (from the function $\mathbb{F}$ and terminal $\mathbb{T}$ sets) in a specific order. There are however, some disadvantages concerning the interpretation of these trees (also known as the genotype-phenotype mapping). However, these are necessary in order to support its more salient features.

These Karva-expressions (or *k*-expressions) are written by Ferreira in the following form [64]:

```
012345678901234567
Q-+/-abaa+a-bbacda
```

The function and terminal sets used in this *k*-expression are the same as those of Section 4.1.1 above, except the terminal set $\mathbb{T}$ now contains the arbitrary constants `a`, `b`, `c` and `d`. The line of digits above the expression is simply an indexing convenience, and the next line is the *genotype* of the candidate.

The sequence of symbols shown is known as a genotype, meaning that is must first be interpreted before it can be used. The interpretation of this (or genotype-phenotype mapping) is shown in Figure 4.5. It is worth noting that neither of the symbols `c` or `d` appear in the phenotype. The phenotype is constructed simply by placing the first symbol at the root of the tree, and then filling arguments of the tree level by level, left to right by advancing through the sequence, symbol by symbol.

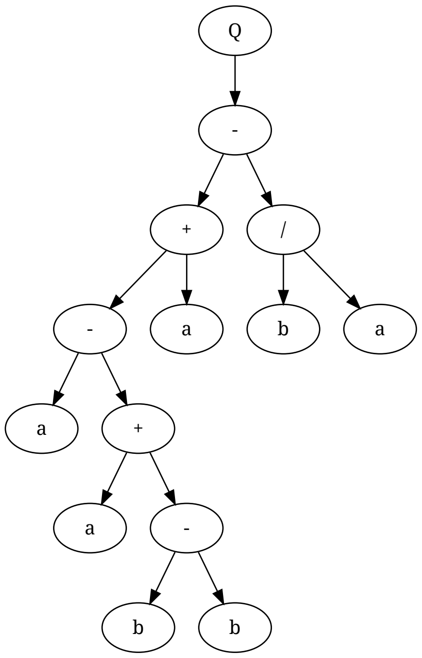


FIGURE 4.5: The abstract syntax tree representing the *karva*-expression `Q-+/-abaa+a-bbacda`.

Although candidates in this form would be kept at a constant length of symbols in the population, the actual depth and construction of the trees depend on the genotype-phenotype mapping, the sequence head length, position and number of function symbols. The symbol string (sometimes known as the *chromosome*) must be divided into head and tail sections, which have their lengths governed by the equation of Ferreira [64] shown in Equation 4.2. In addition, Karva (and hence GEP) can support candidates with any-arity functions, provided this equation holds.

$$t = h(n_{max} - 1) + 1 \tag{4.2}$$

The concept of an intron is one which alludes to the concept of a disabled gene [64]. Candidates may at times have unused symbols in their "genetic code" as indicated by the interpretation of *k*-expressions. These may be activated and deactivated depending on the operators used. This means that potentially useful genes (or symbols) are disabled instead of lost in some cases.

In Equation 4.2, the variable t is the length of the tail section of the *k*-expression, h is the length of the head section, and n_{max} is the largest arity possible in the function set $\mathbb{F}$. Only terminal symbols are allowed in the tail section of the candidate. Should there be too many symbols from the function set $\mathbb{F}$ in the candidate, and not enough terminals, a valid tree cannot be constructed. This is the reason why this equation is necessary.

Simple one-point crossover and point-mutation operators are used in this chapter, in the spirit of the original GP and Genetic Algorithm (GA) [107]. For more sophisticated operators, the reader is referred to the work of Ferreira [64]. As noted in Chapters 8 and 9, there is scope for future work in this area. The additional operators provided by Ferreira are important to ensure the circulation of genetic information throughout a population of candidates [63].

Simple one-point crossover can be applied in a very similar fashion to that of the traditional GA. A random site in a chromosome is chosen, and information is swapped about this point with another chromosome. This generates two new candidates. An advantage of having a linear representation such as *Karva* is that this crossover operator is much simpler to implement than in the usual single-threaded implementation of GP with pointer-trees.

Consider the following *k*-expression:

```
01234567890123456
⋆-+⋆-abaa/abbacda
```

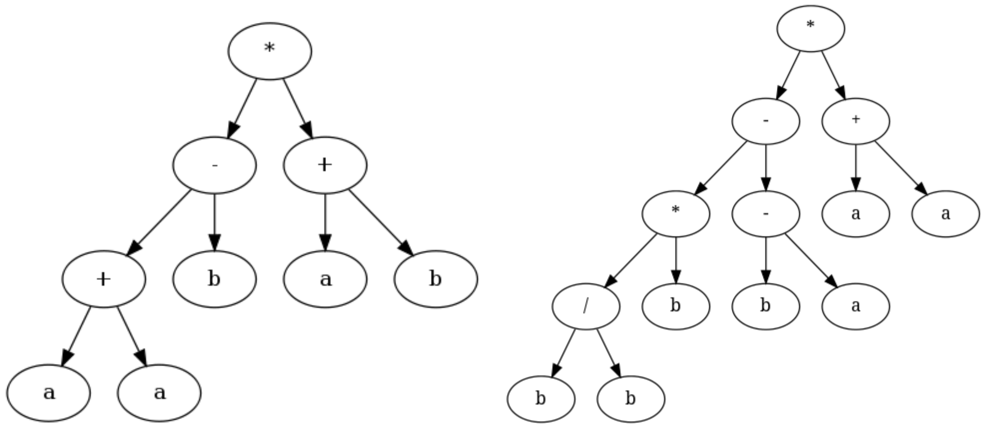
When this expression is recombined with the following *k*-expression, with crossover site at index 5:

```
01234567890123456
⋆-++baa/bbabbacda
```

Then the two candidates generated are shown below:

```
01234567890123456
⋆-++babaa/abbacda
⋆-+⋆-aa/bbabbacda
```

The phenotypes of these new candidates are shown in Figure 4.6. It is notable that the structures in this figure differ greatly, even though the chromosomes are of the same length.



(a) New candidate with genotype `⋆-++babaa/abbacda`.

(b) New candidate with genotype `⋆-+⋆-aa/bbabbacda`.

FIGURE 4.6: New candidates generated by the crossover operator from the "parental" genotypes `⋆-+⋆-abaa/abbacda` and `⋆-++baa/bbabbacda`.

In the traditional representation of solution candidates as pointer tree datastructures, crossover is done by a variety of methods, and most commonly subtree swap/splicing [207, 149]. There is considerable additional overhead traversing a pointer tree in memory (especially on GPU) to find a random subtree and reassigning it to another randomly selected node in another tree.

Point mutation is much simpler. Consider the expression `⋆-+⋆-aa/bbabbacda` whose phenotype is shown in Figure 4.6(a). Assume the head length of this *k*-expression is 8. The symbol `a` at a random index 7, may be swapped for a random nonterminal or terminal, but any symbols in the tail section can only be swapped for terminals. Suppose the symbol at index 7 is switched for the `/` symbol, the expression becomes `⋆-+⋆-a//bbabbacda`

Apart from the considerable lengths necessary to ensure a representation which is suitable for parallelisation, it is also necessary to consider the ramifications of parallelising the genetic operators. Though the fitness evaluation phase is likely to be far more time consuming, it is useful to consider parallelisation of the search algorithm itself when considering the possibility of large populations. The very common Roulette selection method (fitness-proportionate selection) does not parallelise extremely well, for instance. Most parallel GP algorithms make use of a selection method named "Tournament Selection" [183]. In this scheme, every candidate is compared against another uniform-randomly selected candidate, and the best candidate (by fitness) is chosen as one of a crossover pair. Once this process is complete, two candidates at a time are recombined and then the mutation operator is applied.

In the next section, Karva is used as a representation for GP and further considerations for its use on GPU hardware are made. Following this, Geometric Optimisation is introduced in Section 4.5, along with a prominent example of it named the Geometric Particle Swarm Optimiser (PSO) [190, 280], and then a new algorithm in the spirit of the geometric unification is proposed based on the Firefly Algorithm in Section 4.7. Some simple visualisation techniques for these algorithms are presented in Section 4.8.

4.3 Data-parallelism using GPUs

GP and its variants respond reasonably very well to parallelisation. CUDA is used here to improve both fitness evaluation speed and genetic operator speed of the search algorithm itself (including all the operators: selection, crossover and mutation). The CUDA platform arose from a very effective arrangement of MIMD (Multiple-instruction Multiple-data) and SIMD (Single-instruction Multiple-data) processors, which were intended for processing large numbers of pixel data as fast as possible. General-Purpose Graphical Processing Units (GPGPU) has gained much interest since the advent of CUDA, particularly in light of the fact that using pixel and fragment shaders for simulation is an arcane and difficult affair. CUDA is purpose-built for this, and makes this process much more accessible [198]. Additional detail on this is given in Chapters 2 and 3. A short introduction is provided here for ease of reference and self-containment of this chapter. It is recommended that readers already familiar with data-parallelism on Graphical Processing Units should skip this section.

The CUDA-enabled GPU consists of several Streaming Multi-processors (SMs) which have a certain number of "CUDA cores". These SMs process atomic units of work known as "blocks", which represent a 1D, 2D or 3D grid of threads. These blocks are sized by the user, and typically coincide with simulation-specific requirements, such as a grid of threads mapping to every pixel in an image. An SM computes a block until completion, and then, if available, carries on to the next block. During execution, threads are divided into groups of 16, known as "warps". Warps are the smallest unit of execution in CUDA and are executed in a Single-Instruction Multiple-Data (SIMD) fashion on the CUDA cores on each SM, which is sometimes known as Single-Instruction Multiple-Thread (SIMT). The combination of all SMs are therefore MIMD. Additional detail on this is provided in the CUDA programming guide [202].

CUDA-enabled GPUs have some idiosyncratic attributes including memory access penalties and memory scoping among others. These can sometimes be problematic when not given careful consideration. CUDA provides access to a variety of memory banks to the user, each of which has different access penalties and scope. To keep this section brief, a thorough discussion of these is omitted[5]. The process of executing simulations with CUDA involves copying data across the PCI bus to the GPU's global memory, where it is then manipulated by device-specific code. Once this computation is complete on the GPU, the program would copy the modified data back to host memory.

[5]The reader is referred to the wealth of information contained within the CUDA Programming Guide [202].

Depending on applicability in the application, one can make use of host page-locked memory which lessen the effects of these expensive memory copies.

GPU-specific instructions are created by making use of special syntax added to the C language, which is compiled by the nvcc compiler. Once this code has been compiled, the rest of the program is passed to the system C compiler for normal compiling. This allows the CUDA device drivers to insert code to copy device specific instructions to the GPU.

4.4 Karva, Genetic Programming and Parallelisation

As explained in Chapters 3 and 2, the advantages of GPUs are most notably their commodity pricing and high theoretical throughput. Fortunately there is also an inherent parallelism in population-based optimisers. This includes not only parallel fitness evaluation (though it is almost always the most expensive), but also the operators: crossover, mutation and selection. Implementations of GP tend to focus on accelerating fitness evaluations [156], as there is often no need for population sizes large enough to warrant GPU-based genetic operators. Though, if population sizes in these optimisers were large enough, there would an added benefit in using GPUs.

In this section, an algorithm denoted as K-GP-GPU will be presented. It is an adaptation of the original GP to operate on *k*-expressions, which also executes on GPU. It is later used for a base-level comparison.

4.4.1 Parallelisation of K-GP-GPU

From first observations, it appears that *k*-expressions are naturally well-suited to being used with CUDA. This is because they can be stored as sequences of characters or integers and be interpreted. Furthermore, crossover and mutation are almost trivially easy, bearing in mind the head and tail requirements (see Section 4.2). However, a considerable disadvantage is in the genotype-phenotype mapping, which must happen to evaluate fitness. This is discussed in more detail below.

The method used for combining the traditional GP algorithm and Karva is shown in Algorithm 7. The optimisation problem is a modified version of the Santa Fe Ant Trail in $\mathbb{R}^3$ space. Food particles are strewn across a cube with set boundaries, and ants compete to consume the food. The function and terminal sets are the same, except two additional terminals are provided: `Up` and `Down`, which steer the agent upwards or downwards in 3D space.

The majority of computations are parallelised in Algorithm 7. A considerable disadvantage to using *k*-expressions is that the execution of these (or tree traversals) is not straight-forward. Argument-function maps are pre-computed in a separate CUDA kernel in order to allow the interpreting CUDA kernel to directly associate the instructions provided with their corresponding arguments without using recursion. While recursion is permitted in a CUDA kernel, it is generally avoided due to limited stack space.

Precisely the method used to gather inputs and execute the genetic operators depends on the selection mechanism. As mentioned earlier, Tournament selection is the method of choice for the majority of parallel GP implementations. The algorithm used for accomplishing this with the K-GP-GPU algorithm is shown in Algorithm 8.

Probabilities used for determining whether to execute genetic operators on a population for the K-GP-GPU algorithm are $P(\text{mutate}) = 0.1$ and $P(\text{crossover}) = 0.8$.

To facilitate comparison, a single-threaded CPU-based GP optimiser was implemented (which is named here GP-CPU), with the exact same objective function, but computed in a serial fashion using pointer-tree

```
allocate & initialise space for $n$ candidate programs
allocate space for random deviates
**while** termination criteria not met **do**
    call CURAND to fill the random number array with uniform deviates in the range [0,1)
    *copy* candidates and candidate bests to device
    *CUDA*: compute_argument_maps()
    *CUDA*: interpret/execute programs
    *CUDA*: update food locations/fitness
    *copy* back to host
    **if** end-of-generation then **then**
        *CUDA*: apply genetic operators to programs
        replace old programs with new ones
    **end if**
    visualise the result
**end while**
```

ALGORITHM 7: K-GP-GPU: The GPU-parallel implementation of GP using *k*-expressions.

```
launch a CUDA kernel with $n/2$ threads
assign each thread candidate numbers $x/2 + 1$ and $x/2$
set $a$ to random index
**if** candidate $a$ beats candidate $x/2 + 1$ **then**
    replace candidate $x/2 + 1$ with $a$
**end if**
set $b$ to random index
**if** candidate $b$ beats candidate $x/2$ **then**
    replace candidate $x/2$ with $b$
**end if**

recombine candidates $x/2$ and $x/2 + 1$
mutate the two resultant candidates
save results over the original two candidates
```

ALGORITHM 8: Parallel Tournament Selection, with crossover and mutation

datastructures. Crossover and mutation probabilities are the same as the K-GP-GPU algorithm ($P(\text{crossover}) = 0.8$ and $P(\text{mutation}) = 0.01$). The GP-CPU algorithm makes use of the canonical tree-based representation with a depth restriction of 4. The same number of agents were used (1024). Tournament selection is also used, and initialisation/point mutation is done by the *Full* method. Crossover is implemented as a subtree swap. To avoid program bloat in the GP-CPU algorithm, program trees are pruned following the execution of the genetic operators to a maximum depth of 4. Any leaves in the tree with function symbols are replaced by random terminal symbols. It should be noted that this is a very simplistic implementation of GP.

4.4.2 Experimental Results

Some convergence results are provided here for the modified 3D Santa Fe Ant Trail problem, showing how the implementations compare. Performance data is also provided for comparing the GPU/CUDA implementation with a conventional serial CPU implementation (GP-CPU).

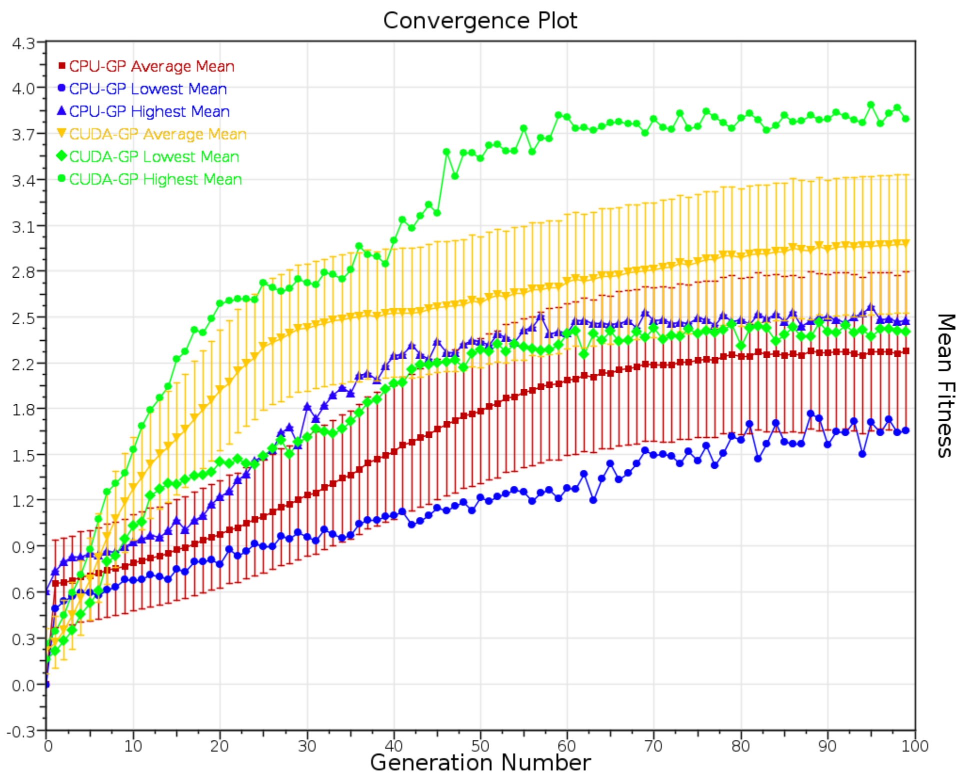


FIGURE 4.7: Convergence results for the K-GP-GPU algorithm with k-expressions and the GP-CPU algorithm (CPU-based GP) with the canonical tree-based representation. The graph shows an averaged mean value of each generation, from 100 independent runs. The error bars represent the average standard deviation of the 100 runs in each generation. Lowest and highest population means are also shown.

Convergence results for K-GP-GPU as well as the GP-CPU algorithm are shown in Figure 4.7. The plot shows the averaged mean values of each generation for both algorithms. Each of these data points have been averaged across 100 independent runs. The error bars on the averaged mean line represents the averaged standard deviation of the population fitness across all 100 separate runs.

It is clear from this graph that K-GP-GPU clearly outperforms the GP-CPU algorithm. At around generation

20, K-GP-GPU seems to have a larger spread in the average mean. It is interesting to note that, because of the difference in representation between the K-GP-GPU (Karva) and the standard GP-CPU algorithms (pointer trees), at generation 20, the standard GP shows the same increase in spread of mean, albeit, less pronounced.

The spread of the mean towards the end of the simulation is smaller for the standard GP-CPU than for the K-GP-GPU. Although a smaller spread is much more desirable, the highest mean of the standard GP only barely surpasses the score of the lowest mean in the K-GP-GPU algorithm. These results should give only a rough indication of the capabilities of these algorithms. Both are likely to be improved considerably given more sophisticated operators and initialisation procedures. Rough convergence properties of Karva as a representation appear to be suitable for further study.

	Frame Time (μ sec)	Gen. Time (μ sec)
K-GP-GPU (k-exp)	1160 ± 40	423.2 ± 0.5
CPU-GP	48000 ± 8300	2600 ± 300

TABLE 4.1: Performance data for the GP-CPU algorithm and the K-GP-GPU algorithm.

Performance data for each of these algorithms were also collected. These are shown in Table 4.1. The timestep compute time was averaged across the 300 steps in each generation, and then across all 100 independent runs. The generation compute time represents the time it took the algorithms to compute a new population only. This was also averaged over the 100 separate runs. The data is of the form mean $\pm$ std. dev.

From this data, the CUDA-based algorithm achieves a speedup of 6 times over the CPU algorithm for computing new populations, and 41 times over the GP-CPU algorithm for computing a single frame of the simulation. A CPU-based GP would not usually be given a population of size 1024, however. Sizes smaller than this is quite likely to be much faster than a GPU, due to a lack of costly memory copies which are necessary to use the GPU instead. The advantages of a larger population are clear when dealing with difficult problems.

Figure 4.8 shows the time taken by each algorithm for computing a new population for fitness evaluation. This process is mostly just the genetic operators; selection, crossover and mutation. It is interesting to note that the generation compute time for the GP-CPU algorithm is nonlinear, while the K-GP-GPU algorithm seems almost practically linear. Given the pointer-tree representation of the GP-CPU algorithm, it is clear why earlier generations are more expensive to generate. Larger pointer trees (generated to 4 levels) require more traversals and memory fetches. Toward later generations, the trees become simpler, and generation of new populations becomes faster for the CPU algorithm.

The mean timestep computing time for the CUDA algorithm has a standard deviation of just 40μsec, whereas the CPU algorithm has a much larger 8300μsec, even taking into account the fact that its timestep computing time is 41 times greater. This is also likely due to the initialisation method (Full) of the CPU-based GP algorithm. Figure 4.9 shows what form a typical initialised agent would take. This is in sharp contrast with Figure 4.10 which depicts a much more effective solution. As can be seen from the initial program, they can potentially contain more `IfFoodAhead` functions, which are far more computationally expensive than terminals. This explains why initialised programs are often more expensive to evaluate.

From observation, it seemed that highly effective programs generally take a certain form. An `IfFoodAhead` function is used to cause movement in a direction other than straight, and the rest of the program consists simply of

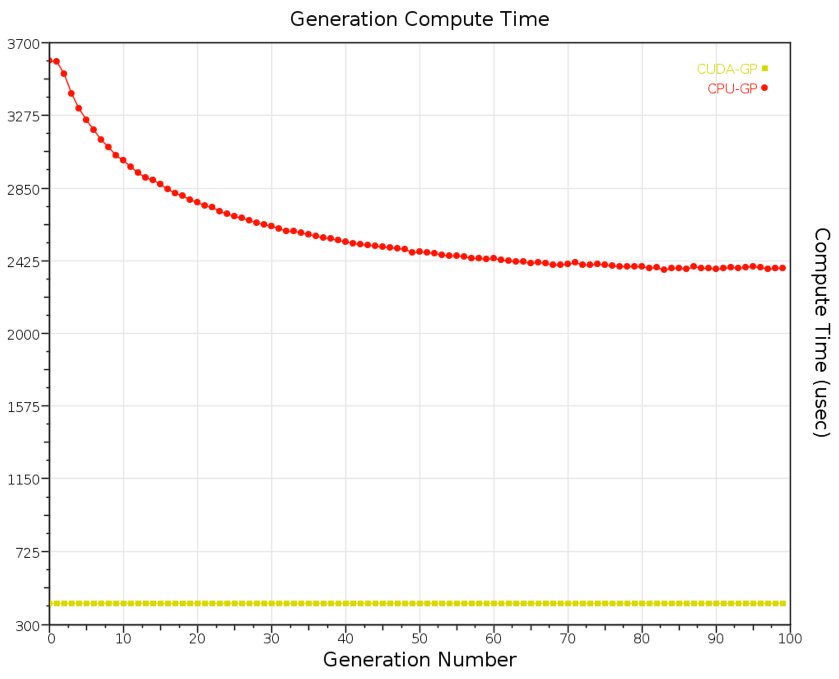


FIGURE 4.8: Computing performance of the K-GP-GPU and GP-CPU algorithms by generation.

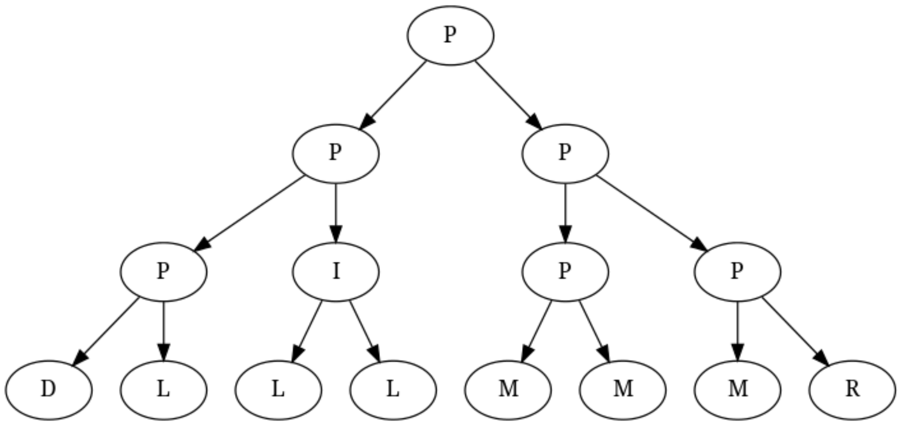


FIGURE 4.9: A typical individual agent generated by the "Full" method in the simulation. LISP-style code for this tree is (P(P(P(D)(L))(I(L)(L)))(P(P(M)(M))(P(M)(R)))).

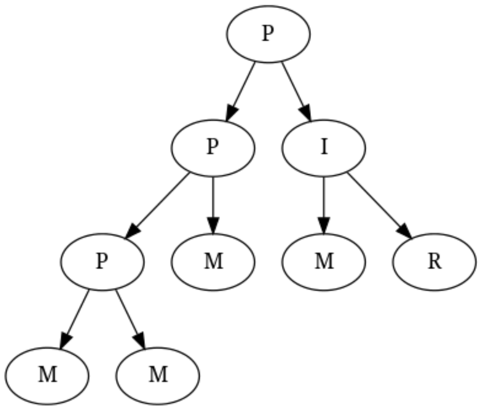


FIGURE 4.10: A highly effective generated agent. The LISP-style code for this is (P(P(P(M)(M))(M))(I(M)(R))).

`Move` terminals. The reason why multiple `Move` terminals are beneficial is because it allows the agent to move faster, and hence potentially reach more food. This modified Santa Fe Ant Trail problem is then perhaps simpler than the original used by Koza [149].

4.5 Geometric Optimisation

Geometric optimisation is a concept that was introduced by Moraglio in his thesis of 2007 [189]. Essentially this methodology brings forth apparatus for converting a parameter-based evolutionary optimiser into an abstract space-invariant optimiser. As is often the case, a specialised optimiser in a particular domain is more effective than an off-the-shelf solution for that domain. Although Poli and colleagues [227] in 2007 noted that it is too early to know if a geometric PSO can compete with other algorithms in the same space as GP. In 2008 Togelius et al. concluded that while their geometric algorithm does not vastly outperform others, it is possible that the PSO is simply not well suited to that particular search space [280].

Despite the initial negative results in the literature, there have been several geometric optimisers proposed. These include Geometric Differential Evolution [192, 191], Geometric Particle Swarm Optimisation [190], and Particle Swarm Programming [280].

The process involving the formal extension of optimisers to other search spaces typically involves redefining the concept of distance [227]; which in continuous global optimisation is a particularly important notion, used primarily for making linear combinations of two or more particles. This also depends on the concrete application intended. Using convex set theory, Moraglio proposed a Geometric version of the PSO [189]. Moraglio also showed that the actual movement of particles is simply dependent on the ability of the algorithm to make linear combinations of vectors in Cartesian coordinate space. Using this, he then postulated that the same can be achieved by using one or more *geometric* crossover operations, which enables detaching the PSO from its originally intended search space. A geometric crossover is essentially a weighted crossover which ensures the intended abstract geometric properties of crossover (ie. keeping the “better genes”, or placing a linear combination of two particles closer to the better particle).

The rest of this chapter is organised as follows. First the Geometric Particle Swarm Optimiser (GPSO) of Moraglio [190] is discussed. Then a new algorithm is proposed named the Geometric Firefly Algorithm (GFA), which makes use of some of the apparatus that Moraglio and Togelius [192, 280] used to generalise algorithms of the same family as the Particle Swarm Optimiser. The generalised GFA is specialised to the search space of programs, and a parallel implementation of it is presented and discussed.

4.6 Geometric Particle Swarm Optimiser

Moraglio, Chio and Poli formally derived the Geometric Particle Swarm Optimiser Algorithm (GPSO) (though without the inertia term) by applying concepts of geometric crossover and geometric mutation using convex set theory [190]. The GPSO is generalised to arbitrary search spaces. From the original PSO, the first observation made is that it must be possible to compute "linear" combinations of more than two particles, in whatever search space is necessary. Moraglio et al. used the concept of a convex combination to ensure that combinations of points in a convex hull in some metric space satisfy four conditions involving the *weight* of the points.

These requirements were given by Moraglio in detail [190], and are loosely reminiscent of the same concepts in continuous space. For example, suppose $\mathbf{g}$ has a fitness (or *weight*) of $w_g = 0.4$, and a particle $\mathbf{x}$ in the space has a fitness (weight) of $w_x = 0.1$, and a linear combination between these are made without randomisation. It follows that the new point n lies somewhere between $\mathbf{x}$ and $\mathbf{g}$, and specifically, more closely to $\mathbf{g}$ due to having a larger weight. The conditions essentially ensure that the weights satisfy $w_x + w_g = 1$, that the point n lies between $\mathbf{x}$ and $\mathbf{g}$, that the distance between $\mathbf{x}$ and $\mathbf{g}$ are coherent with their weights (the mid-point or equilibrium has a weight of zero), and that if $w_g = w_x$ then the distance to the midpoint is the same. More detail on this is available in Moraglio's thesis [189], as well as the work of Moraglio et al. [190].

In order to accomplish the linear combination of several points (ie. multi-parent crossover) in arbitrary space, Moraglio et al. show that it can be done using several separate geometric crossovers [190]. The original PSO, for example, requires the linear combination of a particle with its personal best position found, as well as a global best (ie. a multi-parent recombination). Moraglio et al. also specialised the GPSO to Manhattan and Hamming metric spaces, as well as Euclidean space. Using these ideas, Togelius et al. created the so-called Particle Swarm Programming algorithm, which is a proof of concept optimiser based on the GPSO, and specialised to the search space of genetic programs [280] (of GP). Togelius et al. proposed several possible operators for crossover, and the authors concede that significant research still remains in finding the most appropriate operators. Some effective ones presented by them include weighted subtree swap, weighted homologous crossover and weighted one-point crossover. Homologous crossover ensures that the common region between two candidates are kept intact [226]. Togelius and colleagues reported that common regions can sometimes be too small for this operator to be constructive [280]. The other two operators are more self-explanatory.

4.6.1 GPSO for Karva-expressions

As the authors of Particle Swarm Programming (PSP) have noted, more research would be beneficial for geometric optimisers especially in the search space of programs [280]. By combining the Karva language of Ferreira [64] with the GPSO [190], a different specialisation of the GPSO was obtained. A different search space has a considerable impact on an the ability of an optimiser to explore and converge. This specialisation of the GPSO is presented, and then some performance data is given. The algorithm is parallelised across graphics hardware to improve wall-clock performance. Henceforth this algorithm will be referred to as K-GPSO-GPU, and is described and investigated in this section.

The method used for implementing the K-GPSO-GPU algorithm described above is summarised in Algorithm 9. Parallelisation is achieved in the same manner as the K-GP-GPU algorithm described in Section 4.4, whereby argument maps were pre-computed from k-expressions to allow a separate CUDA kernel to execute programs without recursion.

The weighted crossover of the PSP algorithm of Togelius and colleagues [280] was modified to operate on k-expressions using an adapted multi-parent geometric crossover proposed by the authors of that article, shown in Equation 4.3 [280].

$$\begin{aligned} &\Delta GX((a, w_a), (b, w_b), (c, w_c)) = \\ GX((GX((a, &\frac{w_a}{w_a + w_b}), (b, \frac{w_b}{w_a + w_b})), w_a + w_b), (c, w_c)) \end{aligned} \tag{4.3}$$

```
allocate and initialise enough space for $n$ candidate programs

allocate space for random deviates
while termination criteria not met do
  call CURAND to fill the random number array with uniform deviates in the range [0,1)

  copy candidates and candidate bests to device
  CUDA: compute_argument_maps()
  CUDA: interpret/execute programs
  CUDA: update food locations/fitness
  copy back to host

  if end-of-generation then then
    CUDA: update candidate bests
    CUDA: recombine and mutate programs
    replace old programs with new ones
  end if
  Determine best candidate
  visualise the result
end while
```

ALGORITHM 9: The parallel implementation of the K-GPSO-GPU algorithm.

In Equation 4.3, GX is the crossover operator, and ΔGX is the multi-parent crossover operator. It is assumed that w_a, w_b and w_c are all positive and sum to 1. Essentially this equation defines the weighted, multi-parent crossover as two crossovers, the first being between a and b, where weights are re-normalised to sum to 1, and the second is a crossover with c. Togelius, De Nardi and Moraglio provide more details on this using convex set theory [280]. More detail on the rationale and rigorous mathematical proofs of this procedure are available [192, 189, 190, 280].

As can be seen in Algorithm 9 the majority of computations are parallelised as with the K-GP-GPU algorithm, in anticipation of large candidate populations. In order to execute the programs generated, arguments are again reordered in separate memory space as described in Section 4.4.

At this point, what remains to be determined is precisely how the crossover operation given in Equation 4.3 will be done on linear *k*-expressions. Ferreira defines one-point crossover as choosing a crossover site or "pivot", and then exchanging symbols about this point to obtain two new candidates [63]. In order to ensure that this crossover is geometric in the sense that multi-parent one-point crossover can be computed and still be able to bias the result towards one parent candidate or the other, it must be ensured that it is *weighted*. This is however not enough to define the crossover as geometric. In order to do this, one must prove that this crossover satisfies the four conditions for a convex combination outlined by Moraglio et al. [190]. According to Togelius et al. the following is a reasonable approximation [280].

The method used for accomplishing this recombination is by using the ω, ϕ_g and ϕ_p parameters as the weights (w_a, w_b and w_c) in Equation 4.3. The candidate a is further defined as the current candidate under consideration, b as the corresponding personal best of a, and c as the global best candidate discovered so far. The fitness values of these are not used in the crossover process. Note also that unlike GP, selection is not required, other than simply a value for $P(crossover)$, a probability defined by the user, as in GP. GEP crossover defines a "donor" and a "recipient" tree, which are chosen randomly [280].

The weights mentioned apply to the crossover operation between two *k*-expressions as an indication of how closely to the root of the tree the pivot (crossover site) is. This idea was obtained from the article of Togelius et al. where it was shown that this method roughly satisfies the four conditions of a convex combination [280].

Point mutation proceeds as with the K-GP-GPU algorithm, per Section 4.4.

As the algorithm has been described, focus is now given to an evaluation of effectiveness. The K-GPSO-GPU algorithm described above was compared against the K-GP-GPU algorithm described in detail in Section 4.4. The optimisation problem under consideration was the modified Santa Fe Ant Trail problem in three dimensions described in Section 4.1.2. Analysis involved two aspects of the algorithms, firstly the ability of the algorithms to converge upon a good solution, and secondly the speed with which it was achieved.

The parameters used for the K-GP-GPU algorithm were: P(Crossover) $= 0.8$, P(Mutate) $= 0.1$. The same crossover and mutation rates were used for the K-GPSO-GPU algorithm, and for the PSO-specific settings, these parameters were used: $\omega = 0.1$, $\phi_p = 0.6$, $\phi_g = 0.3$. As for the simulation itself, angular velocities are restricted to 0.1 units, and initial velocities are initialised to between -0.16 and 0.16. This essentially serves to increase the difficulty in finding a good program. In order to use a higher mutation rate, Togelius et al. recommend using elitism [280], whereby the best candidate is replicated verbatim into the new population following the genetic operators. This is a common technique used in EAs to bias the population in a particular direction. Elitism is used in the GP and the K-GPSO-GPU algorithms.

4.6.2 Experimental Results

Figure 4.13 shows the convergence results for the K-GPSO-GPU and the K-GP-GPU algorithms. Each data point in all the plots shown have been averaged 100 times in independent runs. The figure shows apparent evidence that the K-GPSO-GPU algorithm is indeed more able to find a good solution faster, but is quickly superseded by the GP-based algorithm if computing fitness for more than about 25 timesteps. It could be that the GPSO-based algorithm is more appropriate if it is not viable to compute more than 25 generations.

Figures 4.11 and 4.12 show the convergence results for the two algorithms respectively. Elitism was experimented with in order to determine precisely how much of an effect it has on the space of *k*-expressions. Figure 4.13 shows conclusively that elitism allows the algorithms to perform better, albeit marginally.

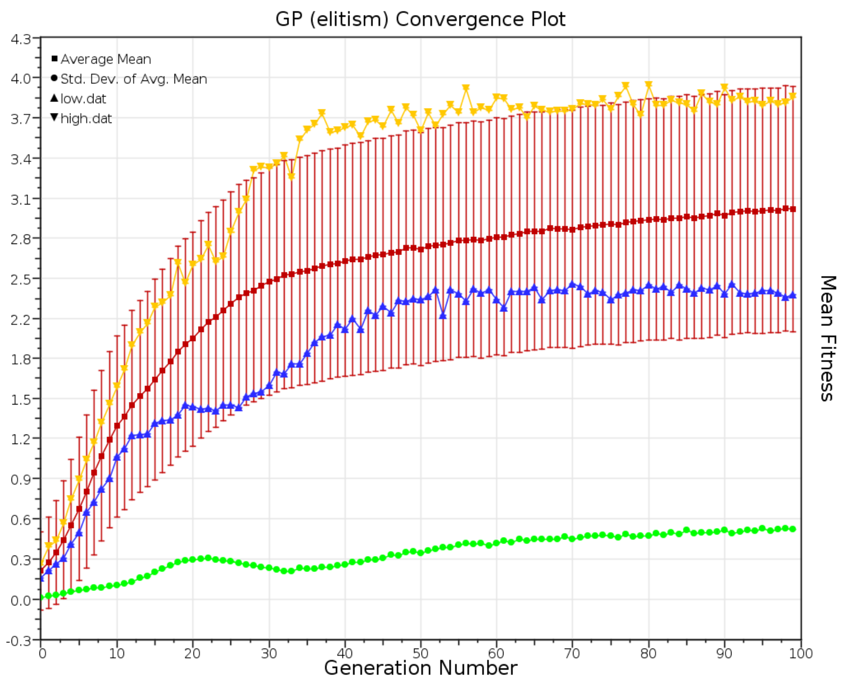


FIGURE 4.11: Modified 3D Santa Fe Ant Trail Problem convergence results for the K-GP-GPU algorithm, with elitism. The graph shows the average mean value of each generation, from 100 independent runs. The error bars represent the average standard deviation of the 100 runs in each generation.

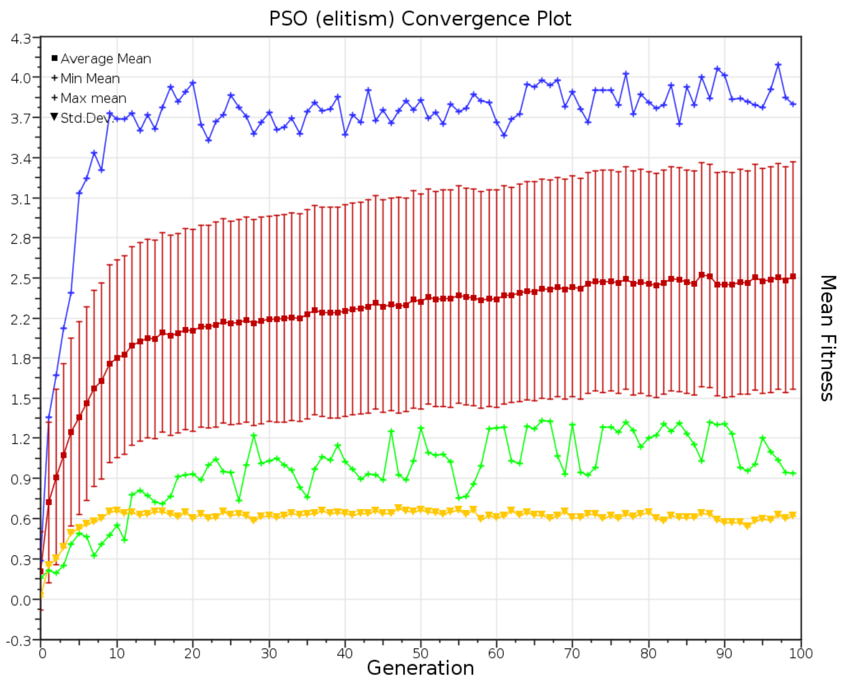


FIGURE 4.12: Modified 3D Santa Fe Ant Trail problem convergence results for the K-GPSO-GPU with elitism. Each data point has been averaged 100 times in independent runs, and the error bars represent the average standard deviation of each generation.

Each data point of the averaged mean was averaged across 100 separate runs. From these plots, it is clear that the K-GPSO-GPU has more spread per generation than the GP-based algorithm, which is not very desirable. Essentially this indicates that the algorithm is somewhat less reliable and not as consistent. The minimum and maximum values are also shown to indicate the full spectrum.

Finally, Figure 4.14 shows the average compute time, by generation, for both the K-GP-GPU and the K-GPSO-GPU. The fitness evaluation consisted of computing 300 frames of the candidate programs and gathering scores from this. Therefore, each data point represents the average frame compute time across each of the 300 frames, and then averaged 100 times by independent runs. The generation compute times are also shown, although they are somewhat hidden. While the first observation seems that the K-GPSO-GPU is faster than the K-GP-GPU algorithm, this is somewhat misleading. Essentially, the plots in Figure 4.14 would be completely linear, if all the terminal and function symbols were of the same complexity. As will be shown later, this is likely due to the lack of diversity in the populations of the K-GPSO-GPU algorithm and the overly quick convergence properties of it.

The average new-generation population parallel compute time for the K-GP-GPU was $420\mu sec$, and for the K-GPSO-GPU it was $440\mu sec$. Even though this is not remotely comparable to the fitness evaluation ($340,000\mu sec$), it was still worth the effort, as this must happen in serial following the fitness evaluation phase. In other words, any improvement on generation compute time will improve wall-clock performance since it cannot be done in parallel with the fitness evaluation phase. This allows to generate larger populations of candidates, but it should be noted that the fitness evaluation phase will also increase greatly in computing time. Therefore, this improved population generation algorithm is likely to be more useful for problems with low-complexity in fitness evaluation.

The function symbol `IfFoodAhead` has a rough complexity of $\mathcal{O}(f)$, where f is the number of food particles, which would approach $\mathcal{O}(fN)$, should all candidates have one of these symbols in its program. Of course, the worst case here is that every candidate consists only of these functions and enough terminals to satisfy the k-expression's head and tail sections. Hypothetically, given a maximum expression length $l = 8$, and a head length $h = 3$ (hence a tail length of 5), then the maximum number of `IfFoodAhead` functions would be 3. Extrapolating from this, assume all N particles were formed like this, then evaluation could reach complexity $\mathcal{O}(3fN)$, which could also very well exceed $\mathcal{O}(N^2)$, a highly undesirable complexity. Of course, this would depend on the value of f, which was arbitrarily chosen in this instance.

Following from this argument, it is possible to make the conjecture that at generation 20, the K-GP-GPU algorithm generally increased its use of the `IfFoodAhead` function, while the K-GPSO-GPU had reached a steady equilibrium of a certain number of these functions. This would seem to agree with the suspicion that the K-GP-GPU is in fact better in preserving population diversity.

Attempts to improve the K-GPSO-GPU beyond the results seen in Figure 4.13 was met with disappointment. Parameter tuning efforts for phi_g, phi_p and ω included normalised combination of respective scores of particles and also normalised weighted scores, but the best parameters observed were simply $phi_g = 0.3, phi_p = 0.6, \omega = 0.1$.

Fine-tuning crossover and mutation probabilities had varying effects on convergence. Removing the crossover phase with the global best solution reduced mean scores to around 0.2, and similar results were obtained from removing the crossover with the personal best. Randomising the crossover point slightly with hand-tuned parameters to aid in diversity did not improve scores at all.

Results indicate that, at the very least, the K-GPSO-GPU operating over k-expressions is appropriate for when the fitness evaluation is extremely computationally expensive. Given enough time and compute power, however, the K-GP-GPU algorithm operating on k-expressions is more suited to this problem.

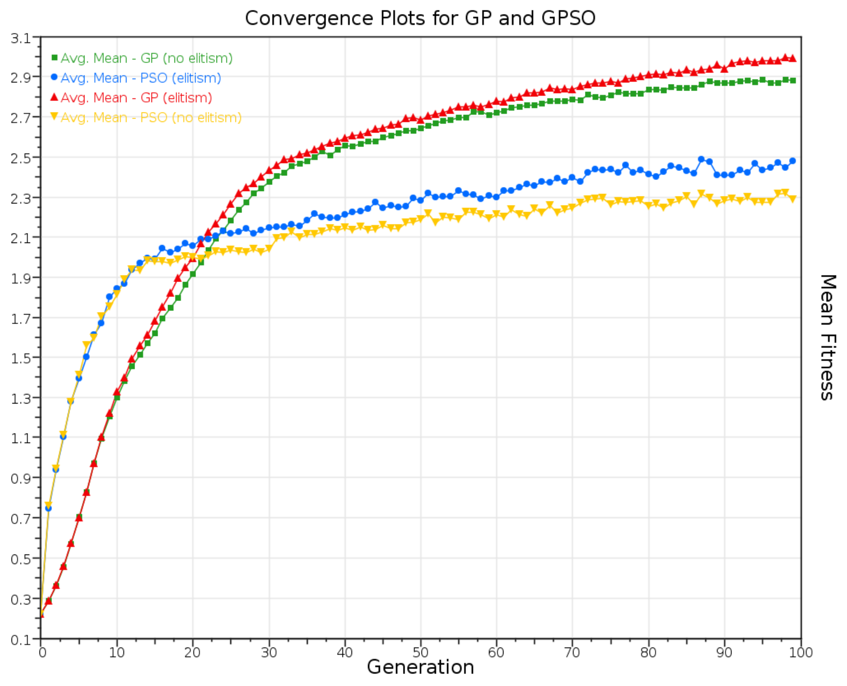


FIGURE 4.13: Convergence results for the K-GP-GPU and K-GPSO-GPU, as well as the use of elitism for both in the modified 3D Santa Fe Ant Trail problem. Each data point has been averaged across 100 independent runs to obtain meaningful statistical data.

As for the performance data presented, it would be unwise to favour the K-GPSO-GPU from the observation in Figure 4.14 that the timestep compute time is lower. The slightly higher compute time does, after all, seem to translate into a higher success rate as shown in Figures 4.11 and 4.13.

Having successfully integrated Karva into parallel modified versions of the GP and the GPSO, a next step is to examine the use of a slightly more sophisticated optimiser, also from the realm of continuous global optimisation.

4.7 Geometric Firefly Algorithm

($\mathcal{I}$, p. 47)

The Firefly Algorithm (FA) is a metaheuristic algorithm from the greater family of stochastic population-based optimisation algorithms for continuous objective functions [299]. It was introduced in Section 3.2.3, and discussed in Section 3.4.2 in the context of parallelism using GPUs. It is now reused and adapted in this section for the search space of syntax trees represented by k-expressions.

In order to prepare for implementing a parallel GFA to operate on the space of syntax trees in the form of k-expressions, multi-parent crossover must be considered. Unlike the K-GPSO-GPU and K-GP-GPU algorithms, the original FA requires all candidates to accept some influence from all others. This therefore requires a “linear” combination of one firefly with n other fireflies in one crossover operation.

Eiben et al. in their work of 1994 [55] and Eiben’s work of 1997 [56] introduced a method known as Gene Scanning for multi-parent recombination. In the context of Memetic Algorithms, there is also some interest in multi-parent crossover operators and Moscato et al. mention multi-parent recombination as a promising area of future study [193]. Eiben introduced several multi-parent crossover operators for binary chromosome Genetic Algorithms. These include uniform scanning, occurrence-based scanning and fitness-based scanning among others.

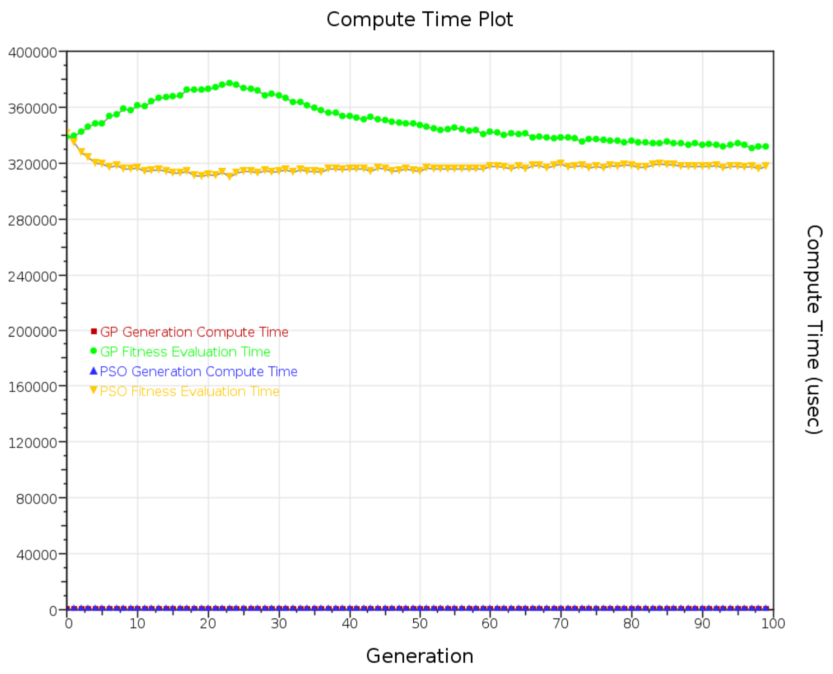


FIGURE 4.14: Modified 3D Santa Fe Ant Trail problem compute times for fitness evaluation, and generation compute time for both the K-GPSO-GPU and K-GP-GPU algorithms.

A trivial modification to Eiben's fitness-based scanning operator is made in order to operate on k-expressions and the symbols they use. The reason fitness-based scanning is used is to ensure that crossover is biased according to the fitness values of each candidate. This allows us to steer clear of adding additional operators[6] in order to ensure the algorithm converges.

The fitness-based scanning operator simply iterates over an expression and selects symbols to copy into the expression from the set of corresponding symbols from all the other candidates. Selecting the symbol for a particular index of the candidate involves choosing a gene with a probability based on the fitness of the candidate from which the symbol is taken. Essentially, the process becomes a fitness-proportional selection operator, which is applied for every expression symbol index separately. More detail on this can be sought from the work of Eiben et al. in 1994 [55]. In order to accomplish a fitness-based scanning crossover operator, it is necessary to use fitness-proportionate selection. This involves keeping track of a running total of scaled probabilities, to which a uniform random deviate can be applied to properly select a certain gene with a probability proportional to fitness. To illustrate how this crossover operator would apply to the K-GP-GPU algorithm discussed in Section 4.4, consider Algorithm 10. Gene Scanning is essentially a replacement for the crossover operator.

Before a FA in the space of programs can be proposed, another issue must be discussed. The original FA update equation is shown below in Equation 4.4 for convenience.

$$x_i \leftarrow x_i + \beta_0 e^{-\gamma r_{ij}^2}(x_j - x_i) + \alpha(d) \tag{4.4}$$

Note that in this equation, distance is necessary in order to compute a light decay function to degrade the perceived fitness of candidates at a distance. Therefore, it is necessary to choose a metric space. For simplicity,

[6]This would effectively turn the algorithm into a Memetic Algorithm [193].

```
allocate & initialise space for $n$ candidates on host and GPU

while termination criteria not met do
  copy candidates to device
  CUDA: compute $k$-expression execution map
  CUDA: evaluate fitness of $k$-expressions with 50 input sets
  copy back to host

  Report averages

  CUDA: apply gene scanning recombination to candidates with P(crossover)
  CUDA: apply mutation to candidates with P(mutation)
  CUDA: replace old candidates with new ones
end while
```

ALGORITHM 10: Parallel Genetic Programming algorithm operating on k-expressions with fitness-based gene scanning.

Hamming distances are used to measure distance between candidates. While in the spirit of the geometric unification of Moraglio, this may not satisfy all the requirements of a geometric crossover, but perhaps only approximately. There are other metrics which may be more suited, such as edit distances, Manhattan distance or fitness difference [26].

Having obtained a crossover operator which is capable of weighted multi-parent recombination in the space of k-expressions using fitness-proportionate selection, it is also important to decay weights with respect to relative distances. Before weights (fitness values) are used, they are degraded by Equation 4.5.

$$s_{b'} = s_b e^{-\gamma H(a,b)^2} \tag{4.5}$$

With a as the current candidate, in this equation, $s_{b'}$ is the updated score of candidate b, s_b is the previous score of candidate b and $H(a, b)$ is the Hamming distance between candidate a and b. Finally, γ defines the shape of the decay curve, which is referred to here as the decay parameter.

With these modifications in place, a fully operational GPU-parallel Firefly Algorithm for expression trees using k-expressions and fitness-based gene scanning for recombination has been created. What remains is to ensure that it performs adequately well, considering the relative increase in complexity from simpler algorithms discussed in previous sections.

4.7.1 Parallelisation

While the fitness evaluation phase remains the same, the process in which candidates are recombined is radically different. Pseudocode at the centre of the k-expression recombination kernel is shown in Listing 4.1. This CUDA kernel is executed ensuring that there is one thread per k-expression. Tiling is used to reduce memory fetch overhead [154, 203, 222] to the same effect as spatial Agent-based Modelling (ABM) as discussed in Chapter 2.

```
__global__ update_fireflies(scores,programs,newprograms,params) {
  const uint index = <global thread index>
  extern __shared__ unsigned char progtile[];
  __shared__ float scoretile[32]; float newscoresum = 0.0f;
  load_my_program(myprog);
  // sum all observed scores for each thread
  for (int i=0; i < agent_count/tile_size; ++i) {
    scoretile[threadIdx.x] = observe_fitness(myprog,
        programs + (i*tile_size + threadIdx.x)*maxlen,
        scores[i*tile_size + threadIdx.x],params);
    __syncthreads();
    for (int i=0; i < tile_size; ++i) newscoresum += scoretile[i];
    __syncthreads();
  }
  float symbolrand[progsize];
  for (int i=0; i < progsize; ++i)
    symbolrand[i]=<rand between 0 and 1> * newscoresum;
  for (int i=0; i < agent_count/tile_size; ++i) {
    globalindex = (i * tile_size + threadIdx.x)*maxlen;
    for (int j=0; j < maxlen; ++j)
      progtile[threadIdx.x*maxlen+j]=programs[globalindex+j];

    scoretile[threadIdx.x] = observe_fitness(myprog,
        programs + (i*tile_size + threadIdx.x)*maxlen,
        scores[i*tile_size + threadIdx.x], params);
    __syncthreads();
    if (<rand between 0 and 1>  < pcrossover)
      for (int j=0; j < tile_size; ++j) {
        running_score += observe_fitness(myprog,
                programs + (i*tile_size + j)*maxlen,scoretile[j], params);
        for (int k=0; k < maxlen; ++k)
          if (symbolrand[k] < running_score) {
            if (scoretile[j] > myscore) myprog[k] = progtile[j*maxlen + k];
            symbolrand[k] = scoresum+1.0f;
          }
      }
    __syncthreads();
  }
  if (<rand between 0 and 1> < pmutate) {
    int pt = <rand int between 0 and maxlen>;
    if (pt < program_head_length) myprog[pt] = <rand int between 0 and 4 inc>;
    else myprog[pt] = 0; // the only terminal (x)
  }
  for (int i=0; i < maxlen; ++i) newprograms[index*maxlen+i] = myprog[i];
}
```

LISTING 4.1: Parallel Geometric Firefly update kernel code. This code includes tiling for scores and programs.

4.7.2 Method

Symbolic Regression (as discussed in Section 4.1.1) is used to evaluate this algorithm against the K-GP-GPU algorithm discussed in Section 4.4. The sextic function is used. The effectiveness of a single candidate expression was evaluated by feeding 50 input/output pairs through the candidate, and if the error in the output is less than 0.01 then it is counted as a successful test. 1.0 is then added to the fitness of the candidate. Essentially this arrangement allows for some noise in the fitness function, since a particular expression tree may be very different from the target, and still obtain a non-zero fitness. The maximum fitness value is 50 for one test. ($\mathcal{I}$, p. 70)

In terms of configuration, candidate *k*-expression sizes were allowed to extend to 64, and head lengths only to 24. The functions symbol set was $\mathbb{F} = +, -, *, \%$ and there was only one terminal symbol, $\mathbb{T} = x$. The search space is therefore comprised of 4^{24} unique *k*-expressions, since in this case there is only one terminal symbol.

Each algorithm was configured with a population size of 1024, mutation probability of $P(\text{mutate}) = 0.08$, and crossover probability $P(\text{crossover}) = 0.99$. For the K-GP-GPU algorithm however, crossover probability was set $P(\text{crossover}) = 0.8$ and mutation probability of $P(\text{mutate}) = 0.1$. These were the best values obtained by empirical testing.

4.7.3 Experimental Results

In this section some generated programs as well as plots of fitness convergence are presented, along with some visual representations of solutions and associated program spaces.

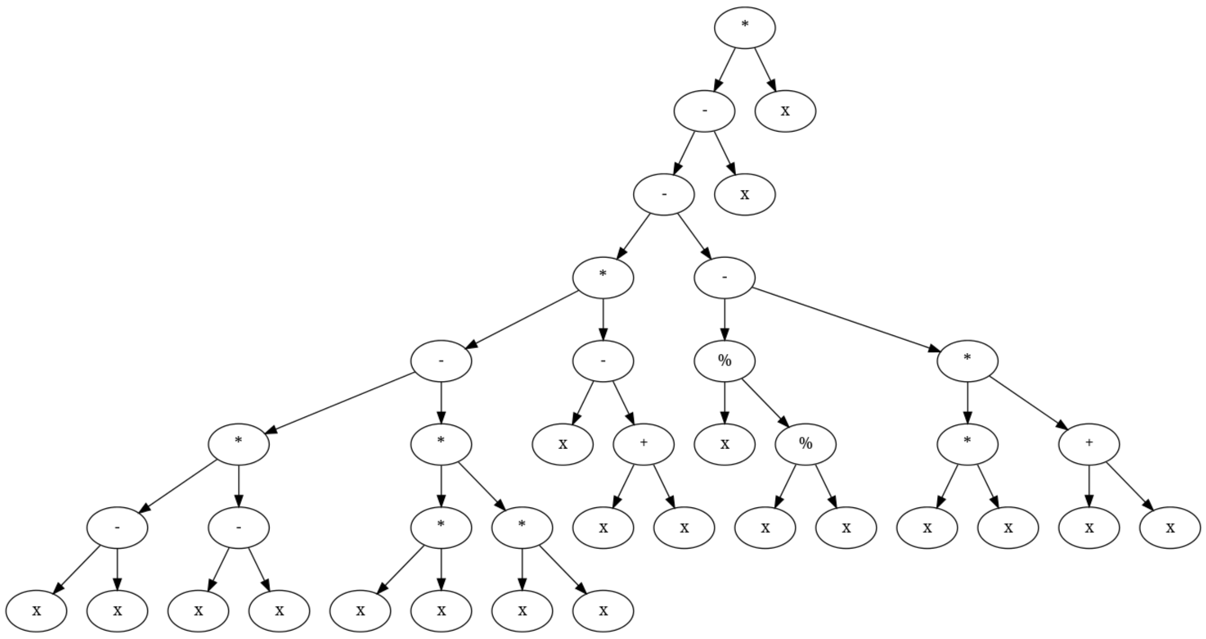


FIGURE 4.15: A sample expression tree generated by the Geometric Firefly Algorithm which yielded a full score of 50 for the Symbolic Regression problem (Sextic function).

Figure 4.15 shows a program generated by the GFA. It resulted in a full score of 50, and when interpreted and reduced, results in $x^6 + 2x^4 - x^2$. This differs in a subtle way from the objective function, in that the $+$ and $-$ operators are swapped. This may well have resulted due to the relatively large error allowed for a successful test with 0.01 in absolute error. Reducing this to a smaller margin of 0.001 may yield a more correct result overall.

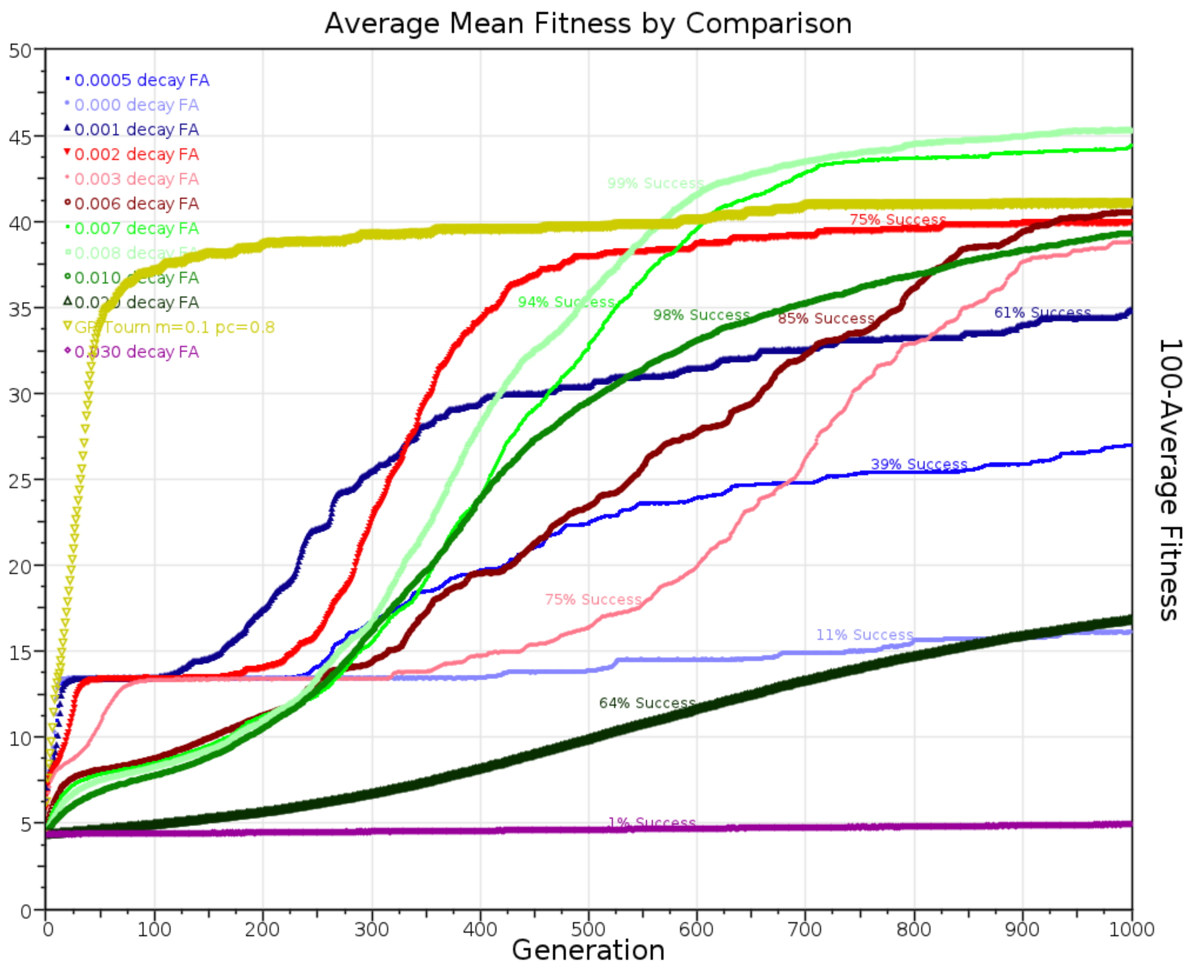


FIGURE 4.16: Comparison plot of different settings of the decay parameter in the algorithm.

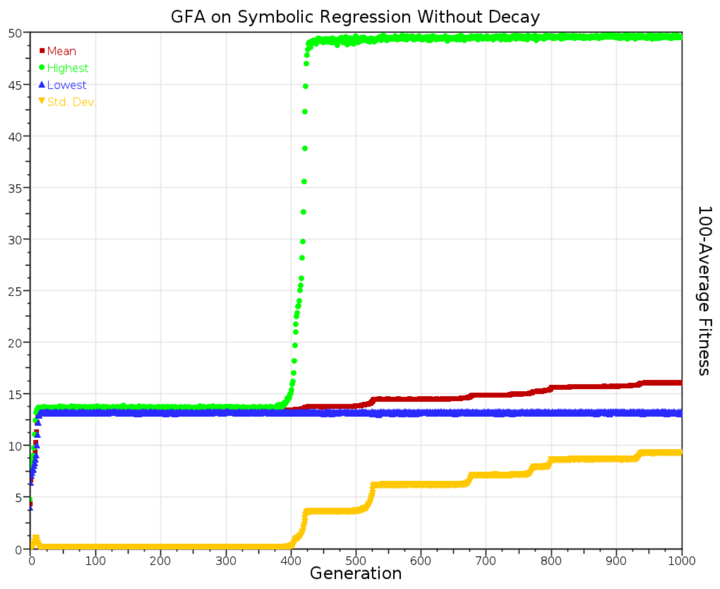


FIGURE 4.17: Fitness plot of the algorithm when observed fitness values from the point of view of one candidate are not decayed at all.

Figure 4.17 presents 100-run averages of standard deviations, lowest and highest population fitness mean, and average population fitness mean. The data plotted in this graph was generated by setting $\gamma = 0.0$ in Equation 4.5, effectively allowing a candidate to perceive the full undegraded fitness values of all other candidates during selection and crossover. For this value of γ, the success rate yielded was an unimpressive 11%.

The plot shown in Figure 4.17 contains some interesting anomalies at generations 420, 530, 680, 760, 800, and 950. The standard deviations appear to climb instantly at these points, and so do the mean, while the lowest and highest remain the same. It is believed that this is caused by the algorithm escaping local minima by pure chance, since the lowest fitness mean is not affected. Essentially, as in continuous global optimisers, this appears to be some kind of premature convergence, where the search would stagnate.

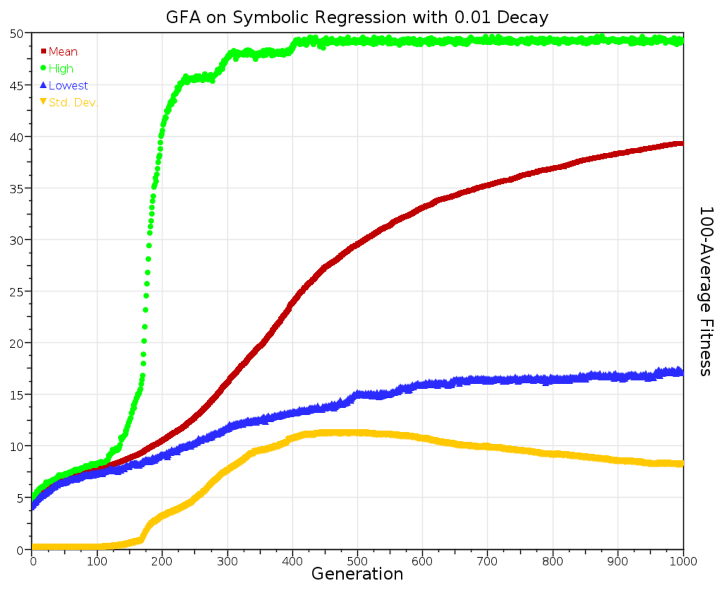


FIGURE 4.18: Fitness plot of the algorithm when the decay parameter is set to 0.01.

Figure 4.18 shows 100-run averages of the highest, lowest and mean population fitness by generation, in the same format as Figure 4.17. Here, the difference is clear, as the average mean increases steadily. For this value of γ, the success rate was 98%, as 98 runs succeeded in producing an expression which produced 50 outputs all within 0.01 of the true sextic function.

Figure 4.16 shows a comparison for different decay values of the algorithm proposed, and also the tournament-selection K-GP-GPU algorithm. This graph is somewhat misleading, as the means do not necessarily convey the success of the algorithm and the range of diversity within them by generation. For example, $\gamma = 0.003$ gains a success rate of 75%, yet $\gamma = 0.02$ which is a similar curve gains a success rate of 98%. Therefore a 3D plot of success rates is provided with mean fitness and generation numbers in Figure 4.19. Here, it is more obvious that similar means obtained do not necessarily correspond to a higher success rate.

Decay curves are shown in Figure 4.20 for various values of γ. For varying values of γ, the observed fitness values of other candidates are decayed differently. The curve with the highest success rate has been highlighted ($\gamma = 0.008$). It is interesting to note, that the Hamming distance between two candidates can reach to a maximum of 64 in this experiment. Even slight changes to the value of γ for this curve makes dramatic changes to the success rate of the algorithm. The effective interaction radius in Hamming space of the algorithm with $\gamma = 0.008$ is

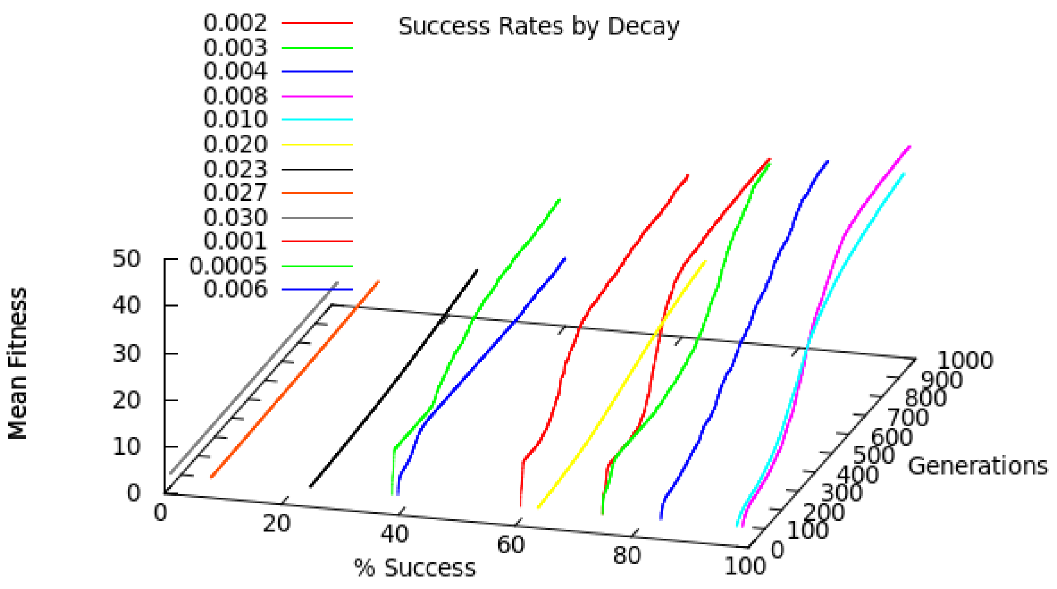


FIGURE 4.19: 3D plot of success rates by decay values for each generation over 100 independent runs for different decay values.

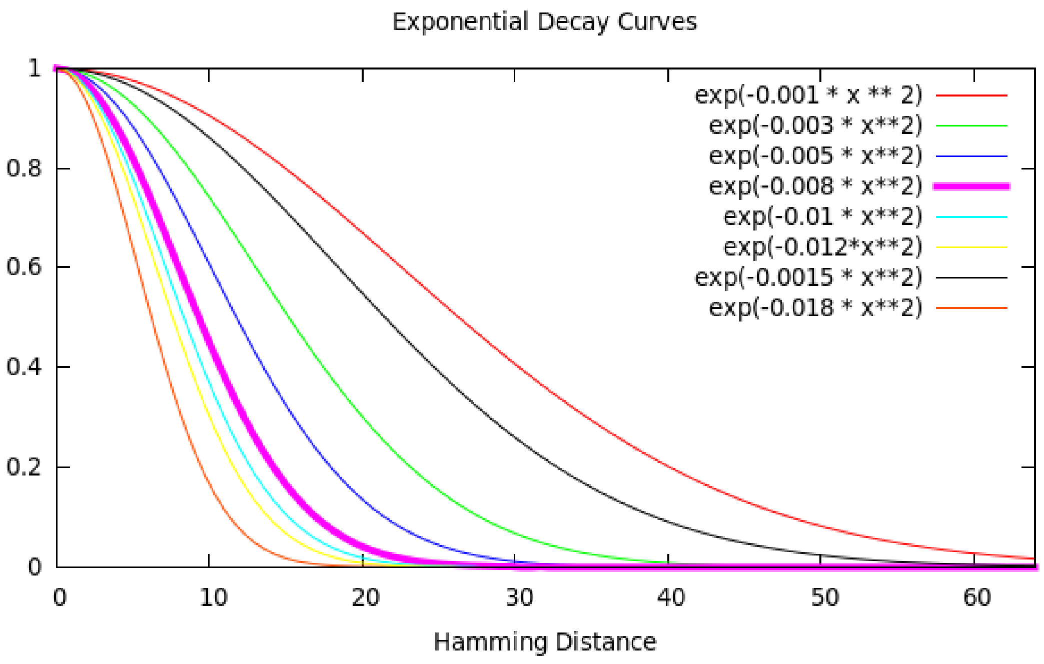


FIGURE 4.20: Sample decay curves produced by various γ values for Hamming distances to candidates observed by the current candidate.

reminiscent of appropriate settings of γ in the canonical Firefly Algorithm, where a γ too large would result in stagnation, and a γ too small would suffer from premature convergence. This particular exponential decay appears to maintain an optimal local search neighbourhood around a candidate.

It appears that suboptimal success rates are achieved if γ is set to not decay observed fitness values at all, or set far too high so as to aggressively decay observed fitnesses.

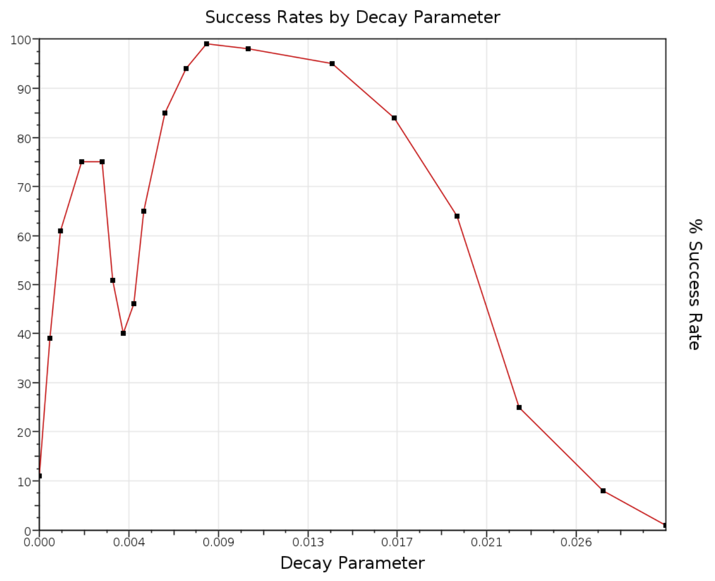


FIGURE 4.21: GFA success rates over the 100 runs of each decay parameter used.

Success rates by γ-value (decay) are provided in Figure 4.21. Given that head lengths are at maximum length of 24, it is useful to look at the shape of the decay curves with γ values between 0.003 and 0.005. What is clear is that the best decay curve appears to degrade observed fitness values greatly if they approach 24. It appears that the best balance in γ is achieved when the curve fits within the head section of the *k*-expressions. It is notable that this may not be the case with other problem domains, however.

Figure 4.22 contains average performance data across the 100 runs of the algorithm. As is expected, initial complexity of the candidates would be high, since the head sections of candidates would be randomly initialised. The result is a decrease in complexity towards generation 500, where average generation evaluation time tends to be around 12 msec. The average generation compute time is at a near constant 31.8 msec. Parallelisation of this algorithm was worth the effort, as it would generally be far more expensive to compute the objective function, therefore the overall compute time was kept in a reasonable range. The outliers in the generation evaluation time is likely due to mutation introducing extra complexity and possibly activating dormant intron function symbols.

4.8 Program-space Visualisation

Visualising the results of algorithms searching through the space of programs not only assists in validation, but also helps in fine tuning and often provides valuable insights. Such techniques have helped cast light on 3D voxel datasets [93], vector fields [117], lattice gases [92] and also spatial datastructures [119] (see Section 2.2.2). They have also been helpful to calibrate and debug algorithms presented in this thesis.

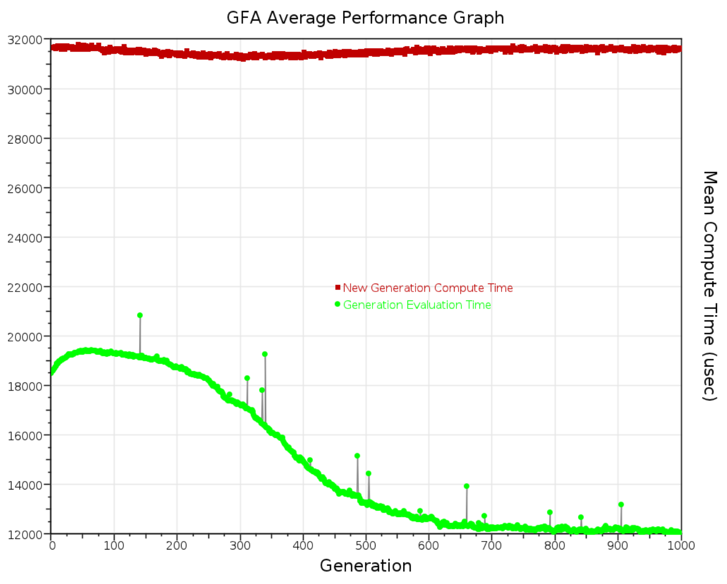


FIGURE 4.22: Average performance data by generation for both generation evaluation time and the time taken to compute a new generation for evaluation.

Two simple methods are presented here to visualise program space. The first is based on the subdivision of 2D space, which works well for shallow-depth programs. The other is a graph based technique, which works better for larger program trees, but is less coherent than the space division method, since two adjacent generations have little in common visually.

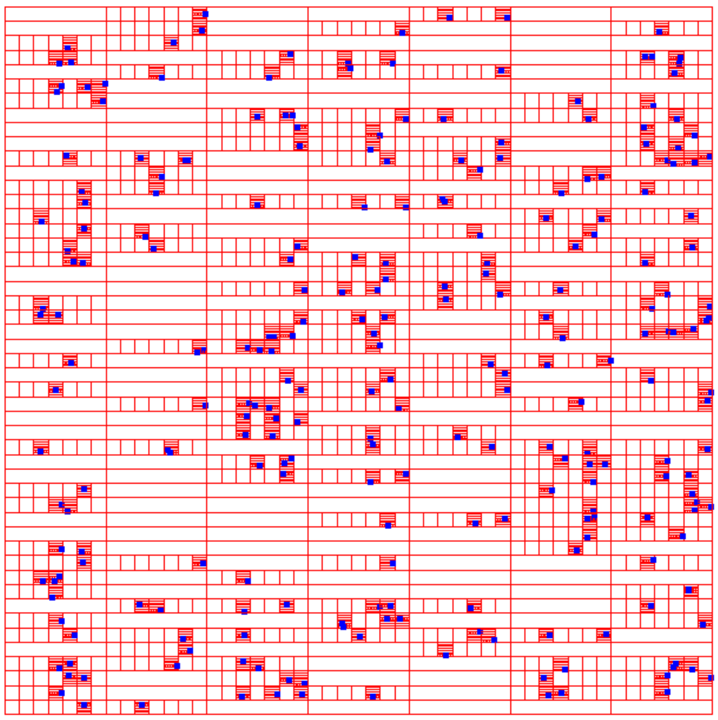
4.8.1 Recursive Space Subdivision

The method used for visualising program space involves a successive subdivision of a 2D grid, where each subdivision represents the selection of a different codon or symbol. This method is specifically engineered for *k*-expressions, but it can easily extend to any other abstract syntax tree representation including pointer trees. This method has previously been discussed by Daida et al. [43], who discontinued its use due to a lack of adherence to widely accepted visualisation principles. The technique is admittedly difficult to interpret when trying to compare individual candidates, and plagued by an excess of lines. However, as an *interactive* tool (with the ability to scale and translate), the author found it useful to diagnose issues with evolutionary optimisers.

Figure 4.23 shows an example of what a randomly initialised population of candidate programs represented by *Karva*-expressions could look like. In this example, a dot represents a single program. The space is divided in a horizontal fashion, for selecting the first symbol, then vertical for the second symbol, and so forth, until all symbols have been selected, at which point a dot is placed. It is worthwhile to note that in doing this, the combinatorial program search space is effectively being viewed as a continuous one (albeit discrete), where differences in programs are represented as spatial differences instead. This idea is essentially equivalent to an edit distance space where symbols towards the root of the expression trees are given a larger weight. In this case, the symbol at the root of the *k*-expression dictates the largest horizontal row in which the expression falls into.

To further illustrate this method, Algorithms 11 and 12 are presented. Algorithm 11 shows the process by which an expression is added to the tree-based data-structure of the visualiser. Algorithm 12 is the method by which the data-structure is drawn to the screen. Algorithms 11 and 12 are kept separate in the implementation, so that interactive use of the program is more streamlined. The data-structure used is similar in concept to k-D trees, where

FIGURE 4.23: Visual representation of a generation of agents using a tree structure.



space is successively subdivided along each of the principal axes, except not binary division, but space is divided into the number of symbols (terminals and nonterminals).

To further align with visualisation techniques in continuous systems, an indication of movement through space is added here. To indicate candidate movement through this pseudo-space in successive generations, a line is drawn from the previous candidate to the new candidate in each generation. This makes certain dynamics of EAs more clear, particularly the *K-GPSO-GPU* algorithm, which is discussed later.

In summary, for a new expression (or program) to be added to the program-space visualisation, the space is first divided into n sections vertically, where n is the number of terminal and non-terminal symbols. Each section represents a symbol. The first symbol in the expression determines the section next divided. Suppose this is the third section from the top (ie. the third symbol in the entire set of symbols, both terminals and functions). This section is then divided into n sections in a horizontal fashion. The next symbol in the expression determines which section will then be divided further vertically, and so forth. Finally, when no symbols remain in the expression, a dot is drawn to indicate the location of the expression. This allows an entire population of expression trees to be shown in a single image.

The visualiser is best used interactively. Keystroke combinations allow the user to scale the image to magnify specific locations within the program space, and translate the viewport to better understand how the algorithm under scrutiny operates. As seen in Figure 4.23, from symbol 4 or 5, it becomes difficult to discern precisely which the final symbols of a program are, but still allows a reasonable comparison among expression trees in a population. The head section of the expression trees tend to be critically important in Karva, and have the ability to greatly change the structure of a tree. This visualiser tends to give top-level symbols more emphasis.

To put this algorithm to the test, a number of visual frames of various algorithms are presented along with a discussion of selected features. In particular, the characteristics of the K-GP-GPU and K-GPSO-GPU algorithms

```
with $n$ candidate programs
with $p$ as the top-level symbol drawable
set $c = p$
**for** $i = 0$ to $n$ **do**
    with $m$ symbols per program
    exp = programs[i]
    **for** $j = 0$ to $m$ **do**
        nextindex = getSymbolIndex(exp[i])
        **if** $p$.children.get(nextindex) is null **then**
            $c$ = $c$.addChild(nextindex)
            $c$.setLabel(exp[i])
        **else**
            $c$ = $c$.children.get(nextindex)
        **end if**
    **end for**
**end for**
```

ALGORITHM 11: Algorithm for adding a *k*-expression to the tree data-structure for visualisation by recursive space division.

```
with $m$ symbols per program

render(top-level)

FUNCTION render(c)
**for** $i = 0$ to $linecount$ **do**
    lines[i].paint()
**end for**
**if** children is not null **then**
    vector2d mystart = getMyStart();
    vector2d myend = getMyEnd();
    **if** orientation == Horizontal **then**
        drawDivisionsHorizontal(mystart,myend)
    **else**
        drawDivisionsVertical(mystart,myend)
    **end if**
    **for** $j = 0$ to childrencount **do**
        render(children[j])
    **end for**
**else**
    drawPoint(mycentre)
**end if**
END
```

ALGORITHM 12: Drawing the tree-based data-structure to the screen recursively.

described in Sections 4.6.1 and 4.4 are compared in terms of convergence for the 3D modified Santa Fe Ant Trail problem. Figure 4.24 show successive generations of the K-GP-GPU algorithm. These figures show that the K-GP-GPU is very effective at maintaining diversity. How this affects the final result will become more clear when the K-GPSO-GPU is discussed.

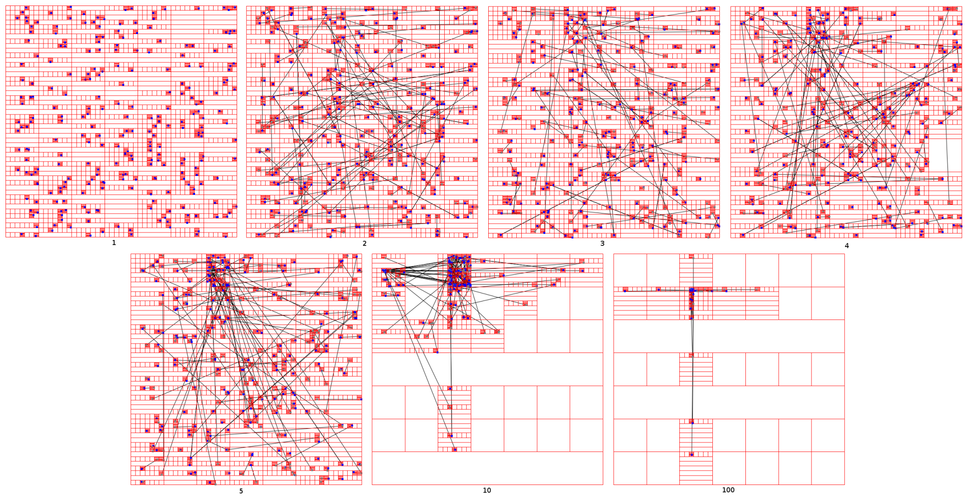


FIGURE 4.24: Populations of programs in generations 1-4 (top), 5, 10, and 100 (bottom) of a sample run of the K-GP-GPU algorithm, showing good diversity with a steady convergence upon the global optimum.

Figure 4.25 shows the second frame of a sample generation from the K-GPSO-GPU optimiser. Immediate impressions that this image conveys is the clear use of a global optimum, which is used in crossover (analogous to the *gBest* particle). It also indicates that there may be an issue in population diversity, as there is little exploration of the top-level symbols. Instead, it seems that an irreversible loss of information is occurring.

To make this more clear, Figure 4.26 shows a plot of the unique candidates by generation for the sample run. Having a good number of unique programs is important to ensure adequate diversity for future crossover operations. The difference in diversity by generation for the K-GP-GPU and K-GPSO-GPU algorithms conclusively indicates that there is a serious lack of diversity in the K-GPSO-GPU algorithm. This could potentially cast light on why the GPSO specialised for expression trees is perhaps not as suitable for the problem of inducing expression trees. Togelius et al. in their work proposing Particle Swarm Programming (the amalgamation of GPSO and expression trees) conceded that it is possible the PSO is simply not well suited to the problem domain [280]. The investigation here on diversity seems to support this, even with the Karva expression tree representation. An improvement upon diversity statistics in the K-GPSO-GPU may bring about a better convergence rate.

Observing the scores from the sample generation of the K-GPSO-GPU revealed a large number of the programs obtained a score of zero. Essentially, in the flow of the algorithm with score-weighted crossover, this would result in a replication of the global best. It seemed that what is necessary is a higher mutation rate. Some empirical experiments were carried out to determine whether calibrating this parameter would improve convergence.

Firstly, a much higher mutation rate of 0.3 is adopted, (as opposed to 0.1), which did not improve the convergence of the algorithm. The standard deviation of the results was too high to be considered a reliable optimiser. Unique diversity in the population was not maintained, since 0.3 still seemed too low. The problem with increasing mutation probability further is that the algorithm would fail to converge at all, as the better solutions would almost certainly be mutated to lower fitness values, by the loss of important information.

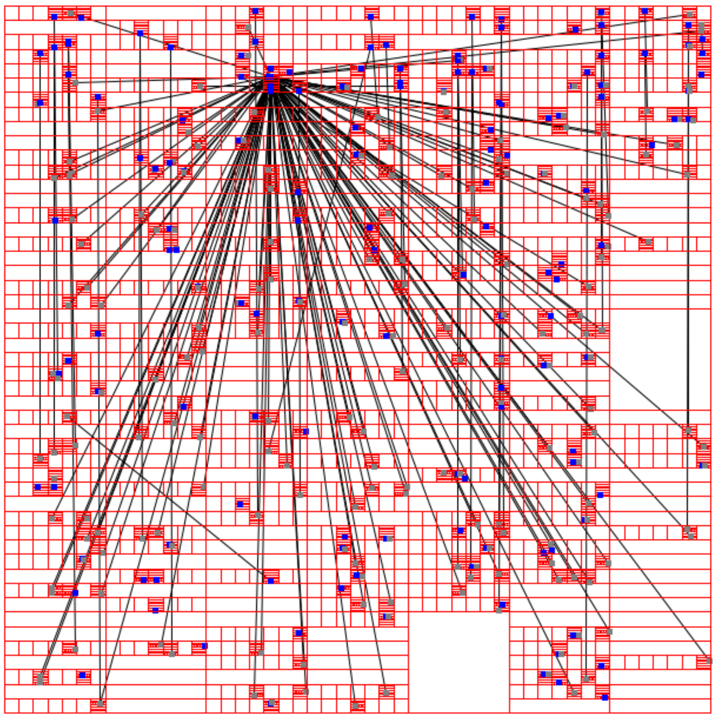


FIGURE 4.25: A visualisation of an early generation in the K-GPSO-GPU optimiser, showing an instant convergence to a potentially suboptimal program expression tree from generation 1 to 2.

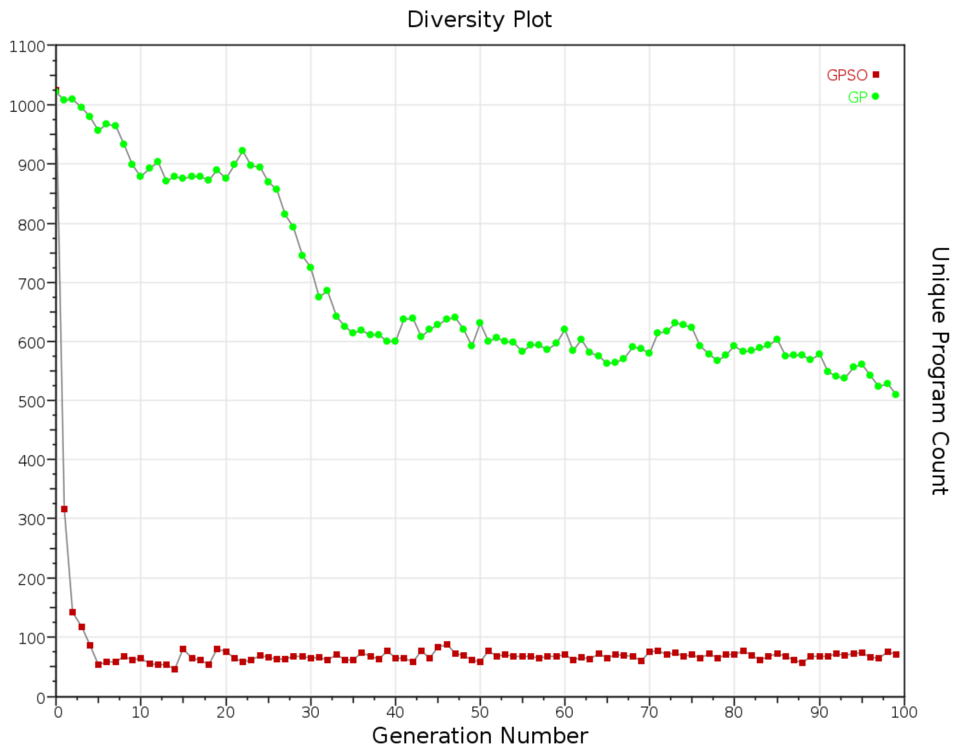


FIGURE 4.26: Diversity plot of the K-GP-GPU and K-GPSO-GPU algorithms during a typical run.

Lowering the crossover rate was also experimented with. This was more fruitful, and resulted in a much lower standard deviation among averaged mean fitness values. A crossover probability of 0.1 seemed to improve the convergence rate. A crossover rate this low does not perform well for genetic algorithms, however. Figure 4.27 shows frame 2 of a sample generation with this modification. In comparison to Figure 4.25, what is clear is that most of the population remains stationary.

FIGURE 4.27: Visualisation of timestep 2 of the K-GPSO-GPU algorithm with modified parameters, at this frame, 907 unique programs are present.

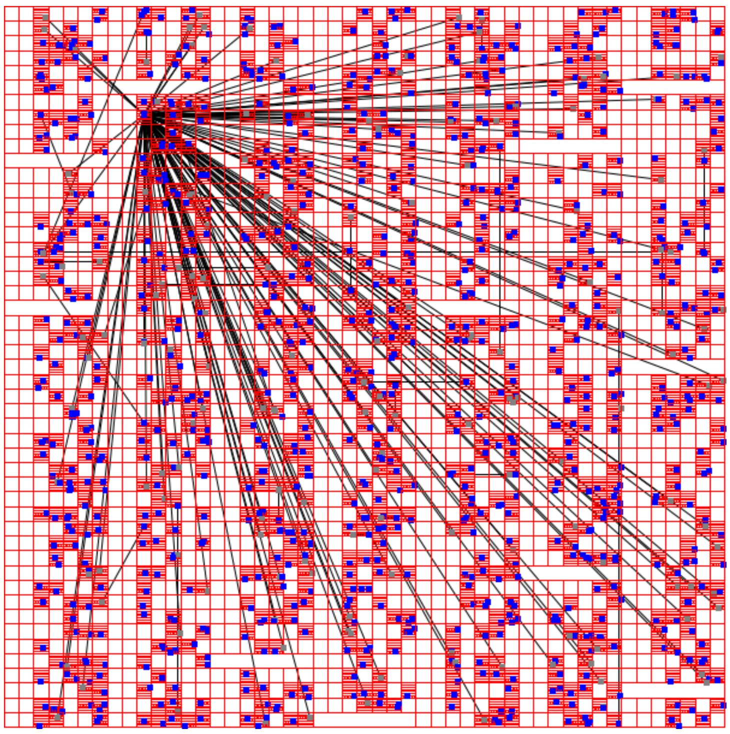


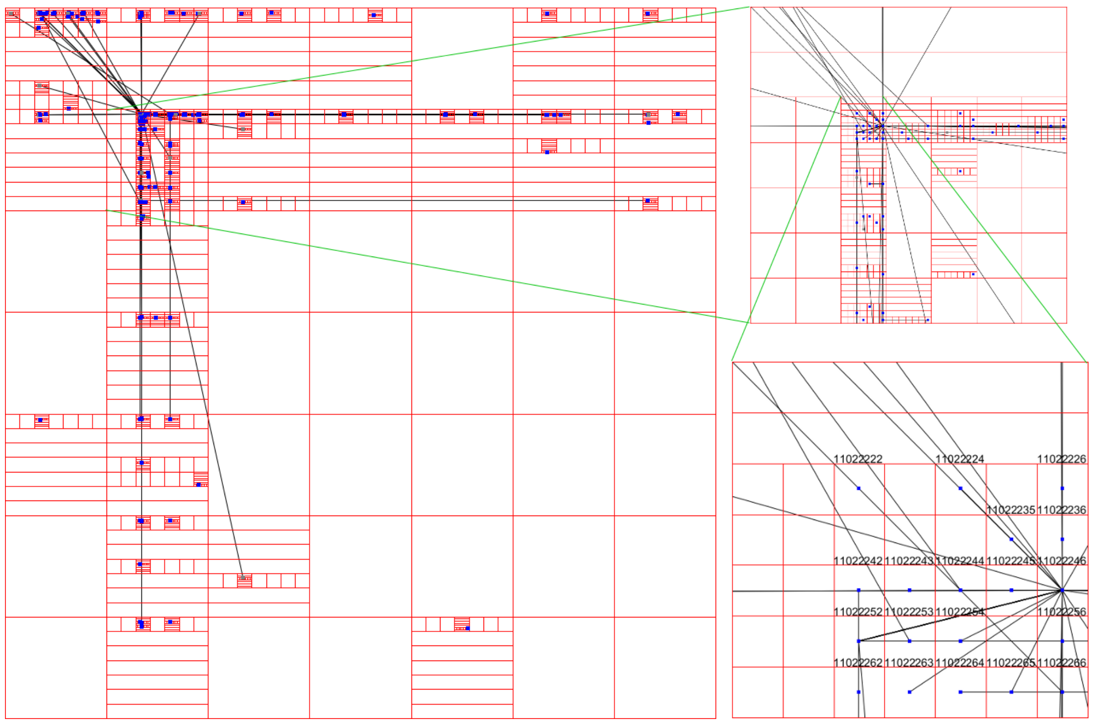
Figure 4.28 conveys a sense of how the visualiser might respond to human interaction. The top-level program space is shown on the left (generation 100 of a sample run of K-GPSO-GPU), and successive scaling in on the area where the most candidate programs are quickly indicates the global best candidate without a doubt.

4.8.2 Graph Rendering Methods

In this section, a simple technique is used to visualise populations of large *k*-expression candidates simultaneously. Such a technique is useful, since the recursive space division technique discussed above becomes less effective when candidates encode more information. Images of large expression trees rendered as a single tree are shown in Figures 4.29, 4.30 and 4.31.

The trees are rendered using a popular graph rendering suite known as GraphViz, particularly the `sfdp` tool for rendering large trees. Populations of *k*-expressions are first pre-processed to construct a graph file. The root node is blank, and the first level of symbols indicate the first symbol of the *k*-expression. The trees shown subsume all expression trees in a population, and re-uses nodes representing symbols that are in the same index location in *k*-expressions. Colours are used to indicate the root (red), progressing through the colours to the leaves (blue). Tracing from the root to a leaf reconstructs an entire *k*-expression. In Figure 4.29 a two dimensional visual representation is given for the first generation of a run with best decay value of 0.008.

FIGURE 4.28: Successive scaling to the location of the global optimum.



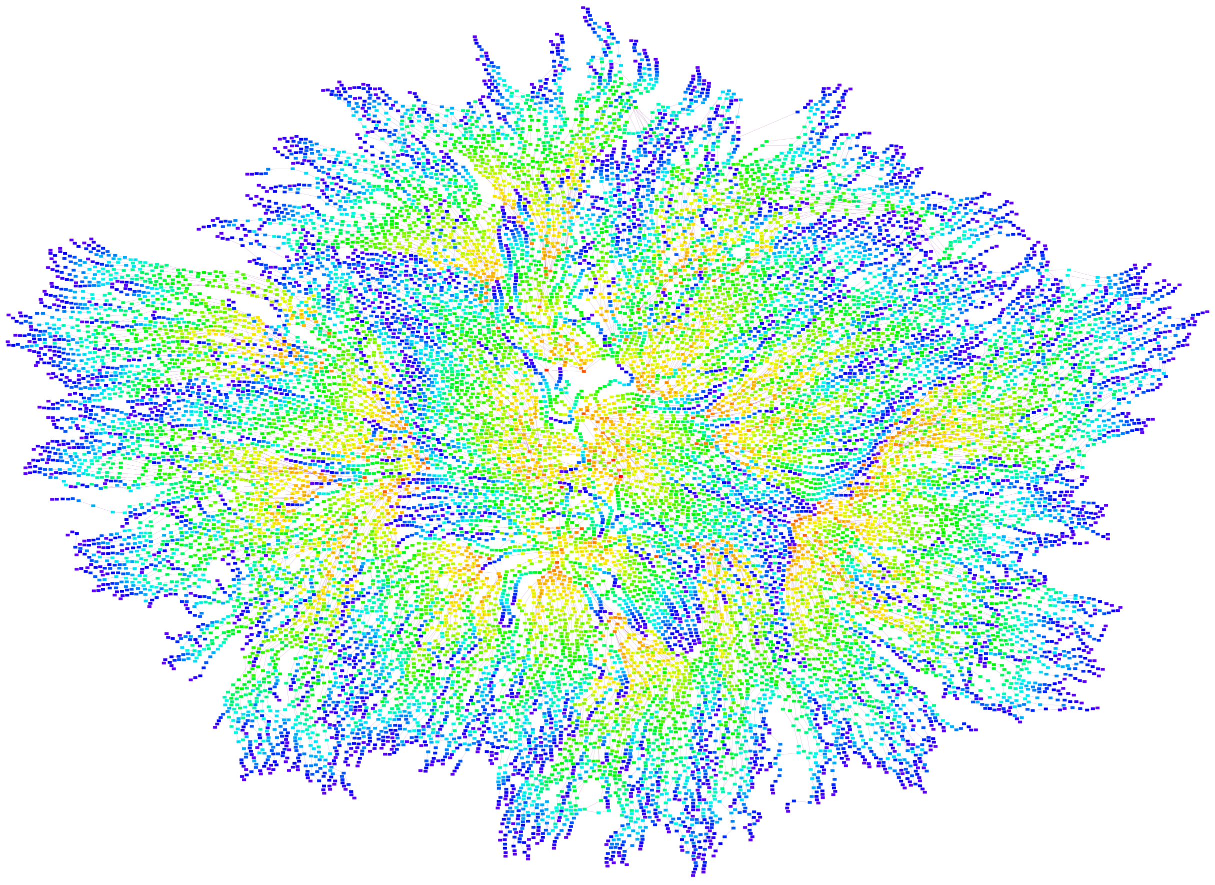


FIGURE 4.29: 2D visualisation of the first generation from the run with the best decay value of 0.008

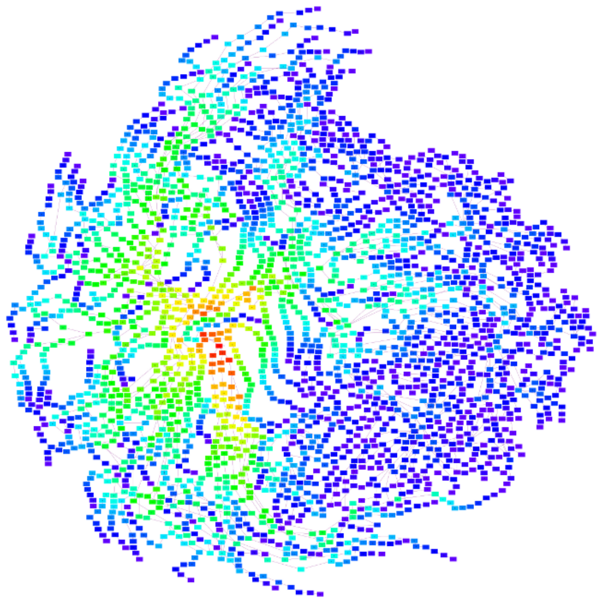


FIGURE 4.30: 2D visualisation of the 500th generation from the run with the best decay value of 0.008

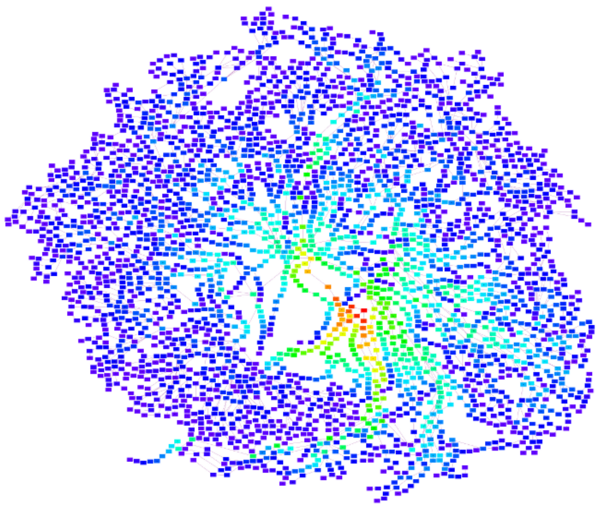


FIGURE 4.31: 2D visualisation of the last generation from the run with the best decay value of 0.008

Figures 4.30 and 4.31 show simpler structures as the algorithm has progressed to converge upon a solution. They show the populations of the 500th and 1000th generation respectively. Features such as bias for a particular strand of symbols is clearly seen, and the number of blue nodes give an indication of the diversity in the generation, while fewer nodes near the root of the tree indicates that there is less diversity in expression head sections.

Part III

MOL: Domain-Specific Language for ABM

CHAPTER 5 DOMAIN SPECIFIC LANGUAGE FOR ABM

Domain-Specific Languages (DSLs) have long been a method of choice for easing the development of software. XML-based markup languages such as HTML and Cascading Style Sheets are prime examples. Agent-based Modelling (ABM) on the other hand, has remained somewhat of an art, where software packages seem either too simple or too complex. DSLs have previously been examined in the context of ABM, and have brought the power of ABM to those in disciplines completely unrelated to computer science. This has greatly increased inter-disciplinary interest and cooperation in the field of ABM.

Though progress has been made, there still exist some shortcomings. Performance is a concern for two reasons. The first being that large systems are either impractical, or completely inaccessible. The second is that temporally distant system configurations are also impractically compute intensive. Improving computing efficiency is therefore important for both these issues. Another shortcoming is that current ABM-related DSLs do not seem to be capable of providing the power that a library can afford, and they are not inherently extensible either.

This chapter and chapter 6 extend upon work previously published by the author in *The 13th IASTED Int. Conf. on Artificial Intelligence and Applications*[1] and the proceedings of the *INMS Postgraduate Conference, Massey University* in 2013[2].

5.1 Introduction

THIS CHAPTER CONSIDERS A NEW EXTENSIBLE DSL for lattice-oriented ABM, focussing carefully on providing as much power as possible while keeping the language simple enough to comprehend easily. The language is also capable of compiling for Graphics hardware or CPU without any additional configuration or syntax changes. This affords both simplicity and high performance concurrently.

Despite the widespread success of ABM in various fields (see Chapter 2), other issues have emerged. Many disciplines do not teach programming as part of their curriculum. It follows that many scientists are not able to leverage the power of ABM [212]. Efforts to mitigate this issue have resulted in a plethora of software packages such as SWARM [186, 276], MASON [169], RePast [38], Netlogo [279], and others[3]. Each of these systems

[1]A. V. Husselmann and K. A. Hawick. Multi-stage high performance, self-optimising domain-specific language for spatial agent-based models. In *The 13th IASTED International Conference on Artificial Intelligence and Applications*, Innsbruck, Austria, February 2014. IASTED

[2]A. V. Husselmann and K. A. Hawick. Towards high performance multi-stage programming for generative agent-based modelling. In *INMS Postgraduate Conference, Massey University*, October 2013

[3]For additional information on these, the reader is referred to the work of Railsback, Lytinen and Jackson [234]

have their own particular merits [234]. NetLogo is of interest particularly because of its built-in DSL to assist the operator [279].

Significant coding efficiency can be achieved through the use of DSLs [99]. Domain-specific knowledge can often carry syntax that is much more clear and concise and therefore ease usage considerably [70]. Though such languages are less general by definition, they serve their specific application domain much more naturally than general purpose languages [182]. Another advantage is that such a language is easy to learn, especially by those who are experts in the domain, but less adept to writing code. These advantages make DSLs very applicable to agent-based modelling platforms. As mentioned above, NetLogo enjoys the use of a DSL [279], and there have been other ABM-related DSLs that have emerged [70, 91].[4]

In recent years, increasing interest in large-scale ABM has created a demand for more research effort, because current methodologies are generally less effective with larger systems [220, 247]. In Microbiology, the sheer number of cells in 100mL of substrate [235] is daunting, and its mass transfer simulation would be extremely expensive to compute [252]. Being able to represent large populations is sometimes a necessity. For example, in ecology, a technique was even developed for approximating the influence of multiple agents in a "super-agent" [252]. In this chapter, focus is given to parallel computing techniques using Graphical Processing Units (GPUs) which will assist in larger scale agent-based models, faster timestep computations, and later, the execution of several agent-based models reinterpreted as objective functions in the realm of optimisation (see Chapter 6). The nature of the architecture discussed here is carefully formulated to allow extension in this direction.

If a simulation is extremely expensive to compute, it will not be practical to compute hundreds of thousands of time steps in reasonable time. Consider the simulation of Boids [240]. If there were 10^7 agents in a simple simulation, the $\mathcal{O}(n^2)$ complexity of a simple implementation would see to it that one timestep is computed perhaps once every ten minutes. To observe time step 10^6 would take more than a decade. For example, it would depend on configuration whether the Boids simulation would exhibit behaviour of interest, and it may not reach equilibrium instantly, or quickly either.

Although NetLogo is programmed by a DSL, it appears to lack certain tools and useful libraries from general purpose languages such as Java and C++ [70]. Effort has recently been expended towards rectifying issues such as these, for example the integration of *R* (a statistical computing suite) into NetLogo [278]. Though the NetLogo DSL leaves much to be desired in flexibility and extensibility, Lytinen and Railsback recommend NetLogo to scientists both new to ABM and those with software programming experience [173]. Disagreements on the usefulness of particular languages such as the Netlogo DSL cast doubt on precisely how applicable these general purpose platforms are in various arbitrary situations.

To date, there have been much research towards the combination of ABM and GPUs. Some of these include applications in medicine [51], other lattice-oriented agent-based models [220] and even marketing [287]. Not many have included the use of a domain-specific language, however. There have been extensive projects in this area for generating parallel code for various other simulations, such as PyCUDA [146]. FLAMEGPU is agent-based however, and transforms agent definitions into parallel code [244].

There have been several DSLs proposed to enhance applications of ABM. Apart from NetLogo [279], BRAHMS is another interpreted language which was initially used for work practice simulation [263], and later applied more generally to social science [264]. There has been limited research in the area of DSLs and parallelism. These particular languages, while interpreted, are not implemented upon a parallel platform such as OpenCL [270] or CUDA [202].

[4]Beal et al. gives a brief overview of several ABM and MAS related DSLs [16].

In the literature it appears that parallel ABM platforms are mostly in the form of a framework, or an interpreted DSL. For example, using FLAMEGPU [244], agents are specified using XML in the form of a specialised finite state machine known as an X-machine [37]. This XML is then transformed using XSLT into compilable GPU-parallel code. Not all of these languages compile to GPU code, however. The CARPET language is built upon an environment which is implemented across a Multiple-Instruction Multiple-Data computer consisting of Transputers [268]. Compilers generate code that is typically much faster than interpreted code, but a great deal of knowledge and time is necessary to implement them, quite often more than interpreters [272]. A paradigm known as Multi-stage Programming (MSP) is discussed below, which can alleviate the difficulties in writing such compilers for DSLs.

Cellular Automata (CA) have some similarities with agent-based models, and have also enjoyed some research with regards to parallel computing. For instance, there has been efforts to parallelise existing CA models written in C to single static assignment representation, then to parallelised OpenMP code [49]. There has also been a language for CAs named CAOS which uses OpenMP [86].

In this chapter, a new language specifically for ABM code-named MOL[5] is proposed. What sets this language apart from others is listed below. The language is built upon Terra, a MSP language [46].

1. It is based on the MSP paradigm
2. Benefits from being a compiled language using Terra
3. Uses LLVM indirectly (through Terra) in order to support run-time code generation
4. Internals of the language are extensible by Terra and C++, and third party libraries
5. Specialising to lattice-based ABM makes the language simple and easy to learn

Two objectives for this language which are important for the work presented in Chapter 6 include machine-code generation for speed, and run-time code generation (RTCG). This is to ensure that code can be modified at run time as necessary, and still be executed quickly.

The rest of this chapter is organised as follows. Firstly, a short introduction to the Terra language and the MSP paradigm is provided in Section 5.2, followed by details in Section 5.3 on the proposed new DSL compiler. Some selected models are briefly used to demonstrate the utility of this new language in Section 5.4.

5.2 Multi-stage Programming and Terra

The language developed in this chapter is itself built using Terra [46], a very recent low-level extension of the Lua language [131]. It is a fast language which compiles to memory just-in-time (JIT) before being executed on a stack separate to Lua [131, 130]. Terra incorporates the notion of MSP, which is a technique that specialises code for when data arrives in “stages” [274, 275]. It does so by using the LLVM [159] internal representation, compiling code assembled at run-time into machine code using standard backends to LLVM. Although Terra is not the only language which is capable of run-time code generation (RTCG), it was chosen for its elegance and simplicity. Other languages which are also capable of RTCG include MetaOCAML [42] and PyCUDA [146]. PyCUDA is based on the notion of building programs by constructing a string [146], which is generally considered less effective [274] than more integrated approaches such as MetaML.

[5]MOL is not to be confused with *many-optimising-liaisons* in MOLPSO.

There are a few advantages to using multi-stage programs, including efficiency, productivity and quality [274]. To illustrate the first advantage, Taha gives the example of a power function, which can raise its input to any exponent by recursion [272]. The idea in this example was to point out that if the power of 2 is frequently evaluated, then it could sometimes be useful to define it separately. Should there be other powers that are computed, then MSP allows to automatically generate these. Another example given by Taha and Sheard is that of matrix multiplication [275]. Should the information required to compute an inner product of two vectors arrive in different time frames, such as the size of the vectors, the first vector, and then the second vector, different optimisations can be done for each stage. This concept was even used by Reinholtz in 2000 to postulate that the Java language will eventually *exceed* C++ in performance due to Just-in-time (JIT) run-time compiling of code using additional information available to the compiler to make further optimisations [238].

Increased productivity from MSP can be achieved by storing problem-specific information in the form of a program generator using MSP [274]. Program generators are capable of much more than simple macro expansion. Taha also indicates that this gives increased reliability, because a program could have the capability to adapt itself, and therefore not require error-prone human intervention [274].

The Terra language makes full use of the MSP paradigm and also conveniently integrates with Lua. So much so that the lexical environment of Lua is made available to Terra-specific code [46]. An example of the ubiquitous "Hello World" program in Terra is given in Listing 5.1.

```
C = terralib.includec("stdio.h")
terra helloworld(argc: int, argv: &rawstring)
    C.printf("Hello, world.\n")
end
helloworld(0, nil)
terralib.saveobj("hello", {main=helloworld})
```

LISTING 5.1: The "Hello World" example in Terra.

Scripts are interpreted as Lua, until a terra symbol is encountered. A Terra function is then parsed, and stored as an untyped tree until it is called in the same style as a Lua function. At this point, the function is JIT compiled to memory, and executed. In fact, the script files in which Terra is used are JIT compiled by LuaJIT [209], which is a complete reimplementation of Lua.

Terra uses Clang to provide some C++ interoperability, and in this case, the C printf function was used. In addition to this, terralib saves the compiled Terra function to the "hello" executable. saveobj requires the correct signature in the Terra function declaration, however. Terra functions are not restricted to a certain set of arguments.

Listing 5.2 gives an example of how a power function might be specialised to increase performance and reusability. Here, the Lua function make_power_func takes an integer exponent and uses an inner recursive Lua function to generate an expression which will compute the required power of the abstract symbol x. Finally, the expression is spliced into a Terra statement within a Terra function, and the function is returned. A useful property of Terra functions is that they are first-class Lua values, and can be passed from function to function as variables.[6]

The essence of the architecture presented in the next section effectively relies on the proof-of-concept code shown in Listing 5.3. This code is entirely valid Lua-Terra, except for the second line mol move towards ex a done is invalid syntax, made valid by extending the Lua lexer and parser with a separate script using a Pratt parser

[6]For a more thorough introduction to Terra, the reader is referred to the work of DeVito and colleagues [46].

```
function make_power_func(int_exponent)
    local x = symbol(int)
    local list = terralib.newlist()

    local function rec(i)
        if i == 1 then return x end
        return `x*[rec(i-1)]
    end

    return terra([x]): int
        return [ rec(int_exponent) ]
    end
end

local test = make_power_func(3)
print(test(2)) -- output is 8.
```

LISTING 5.2: An example of staging using Terra. The end result is a function which creates a JIT-compiled power function.

```
a = {0.1,0.2,0.3}
tempf = mol move towards ex a done
tempfunc = tempf.impl

agent = {}
agent.step = terra(all_pos : &float,   all_vel : &float,
            pos: &vector(float,3),  vel: &vector(float,3), agent_count: int)
   var t = tempfunc()
   @vel = @vel + (t - @pos) * 0.001
   @pos = @pos + @vel * 0.1
end

terra stepAll(positions : &float, velocities : &float)
   agent.setall_pos(positions);
   agent.setall_vel(velocities);
   for i = 0, [ agent_count ] do
      [ agent.step ](positions, velocities,
            [&vector(float,3)](positions+3*i),
            [&vector(float,3)](velocities+3*i),
            [ agent_count ]
        )
   end
end

stepAll:compile()
parameters.setStepallFunc(getFunctionPointer(stepAll))
```

LISTING 5.3: Proof of concept Lua-Terra code for exposing a JIT-compiled Terra function to compiled host code.

[228] provided as a library extending the Terra parser. The same line is actually JIT compiled as well, using the Terra `quote` operator to combine statements together. The grammar of this short code segment is very simple, and extensions are described in the next section. The rest of the code in this listing is library code. The stepAll() function is a Terra function which maps the agent.step() function to all agents in the simulation, splicing in a variable from Lua named agent_count, as well as the Lua table containing the step function. The variables position and velocities are host-allocated memory segments.

Having established some advantages of MSP and Terra, it is perhaps more clear why its use as a DSL constructor would be desirable in the context of ABM. The rest of the chapter is dedicated to presenting and giving details on this new DSL for lattice-oriented ABM, including its implementation and examples of its use.

5.3 The MOL Compiler

Using Terra and making use of the advantages of MSP in a fully custom language is not entirely trivial. A special separate compiler architecture is needed, which constructs Terra statements and functions using multi-stage operators. In this instance, this additional compiler architecture includes a parser, a type checker, and a code generator. The Terra libraries provide good support for writing these in Lua. The addition of dynamic typing and garbage collection in Lua allows for simpler tree structures and simpler traversals.

5.3.1 Lattice-oriented ABM

At this point, the language caters for lattice-oriented agent-based models only, where agents are homogeneous. That is to say that agents are identical, and therefore execute the same code. They are, however, not restricted to exactly the same code flow. The final objective of the compiler is to generate a valid and semantically correct Terra function which conforms to the following signature:

```
terra( rolattice : &int, wolattice : &int,scores: &double,frame: long).
```

This permits the host program to maintain the lattice itself and visualise the results in real time. It is not necessary to force the host program to maintain lattices, however. Terra is capable of using C functions to allocate memory very simply. In this instance, it was chosen to allocate and maintain these in the host program in order to simplify rendering.

Figure 5.1 indicates the flow that the DSL framework would follow during startup. When the Lua parser encounters a `mol` token, the special MOL parser will process this syntax, then hand a parse tree to a type checker, and finally the type-checked parse tree is passed to a code generator. During the code generation phase, the compiler will emit Terra statements and expressions, which are combined together to form a Terra function with the signature given above, which is then JIT-compiled.

MOL is equipped with an extensible custom library, which makes available common functionality in lattice-based simulations. Features such as random numbers and movement are provided in this library. This also presents an opportunity to specialise code for the benefit of the simulation, as library functionality is made available in the form of Terra statements and expression generators. Thanks to meta-methods and Terra structures, data types such as 3D integer vectors and 3D floating point vectors can support various binary and unary operators. These structures can be used within the DSL described here quite simply, due to the shared lexical environment of Lua and Terra. Introducing other structures would be almost trivial.

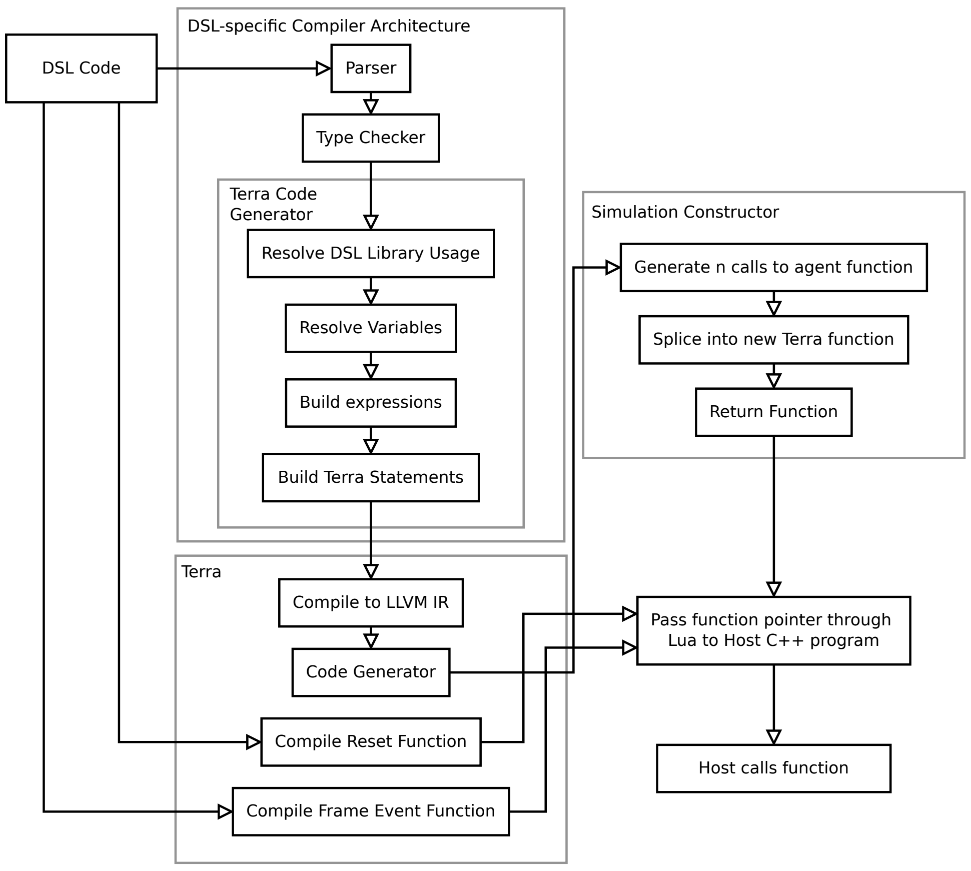


FIGURE 5.1: A flow diagram indicating the initial program flow of the DSL framework presented here.

Following the code generation phase, Terra statements are then passed to Terra, which compiles the syntax tree to the internal representation (IR) of LLVM [159]. This is then compiled using one of the standard LLVM backends for the X86 instruction set (in this case). At this point however, the process is not complete. What has been compiled to machine code is a function with the following signature:

```
terra( rolattice : &int,wolattice: &int,me: int, score: &double, x:int , y:int , z:int)
```

When called with the correct arguments, this function will execute the set of agent instructions for precisely one agent. What is needed is to construct another Terra function which calls *this* function n times with the correct arguments each time in order to cover the entire lattice. This is indicated as the "simulation constructor" in Figure 5.1. For n lattice sites, a call to this function is generated and spliced into a list of Terra statements. Listing 5.4 contains a short code excerpt showing how a Terra function is constructed for computing the same agent behavioural code on all lattice sites. Essentially, it is possible to change this framework into a heterogeneous agent-based simulation platform by either differentiating between different types of agents in the DSL code, or linking different programs within this code.

```
local myprogram = molcompiler.compile(prototype.typedtree)
local temp = terra([ rolattice ], [wolattice], [scores])
   for x = 0, [ sim_grid_x ] do
      for y = 0, [ sim_grid_y ] do
         for z = 0, [ sim_grid_z ] do
            [myprogram](rolattice, wolattice,
               rolattice [ z*[sim_grid_x]*[sim_grid_y] + y*[sim_grid_x] + x],
               [&double](scores), x,y,z
            )
         end
      end
   end
end
temp:compile()
parameters.setStepallFunc(getFunctionPointer(temp))
```

LISTING 5.4: Lua-Terra code excerpt showing how a Terra function is constructed to execute the same agent program on all lattice sites.

The escape operator [ ] is used to splice Lua values into Terra code. These may include Terra statements or expressions and these are evaluated at compile time (therefore Lua values are essentially constant). More details on this is available [46]. Once the function is constructed, it is compiled and the function pointer is passed back to the host program using a Lua function. Along with this function which will compute an entire time step given the lattice arrays from the host, two other functions are also compiled and passed to the host program. These functions facilitate a lattice reset and an extra function to execute on every timestep computation. Additional computation during frames can be costly, and therefore, the frame event can be disabled easily. This is useful for gathering data into separate files, and the reset function is simply used to initialise the lattice.

As an example of the process of compiling staged code, the example shown in Listing 5.2 is specialised into the code shown in Listing 5.5. This code is then compiled into the LLVM IR code shown in Listing 5.6. Finally, this IR code is then compiled using a backend of LLVM, which in this case would be the X86 backend.

The parser, type checker and the code generator for MOL are implemented in a recursive manner, which are highly influenced by the example code provided with Terra [46]. Syntax trees (type-checked and otherwise) are

```
__terratest_t_14_0  = terra($x : int32) : {int32}
   return $x * $x * $x
end
```

LISTING 5.5: The Terra code generated by the staging example shown in Listing 5.2. The result is a compiled function which computes the cube of an input integer.

```
define i32 @__terratest_t_14_0(i32) {
entry:
  %1 = mul i32 %0, %0
  %2 = mul i32 %1, %0
  ret i32 %2
}
```

LISTING 5.6: The LLVM IR code generated by the staging example shown in Listing 5.2. This code is the final stage before a LLVM backend compiles the code to machine instructions.

constructed using Lua tables, making full advantage of the fact that Terra statements, expressions and functions are all first-class Lua values.

5.3.2 Spatial ABM

Though the emphasis in this chapter is given to lattice-oriented ABM, a spatial version of the MOL language was also experimented with. It is easier to see the efficacy of MOL on lattice-based models, but it is certainly possible to cater for spatial ABM also, and this will be briefly detailed here.

Where the compiler would generate an agent Terra function which takes `int`-lattices, the spatial version generates an agent function with the following signature:

```
terra(positions: &float, velocities : &float,scores: &float,frame: long)
```

In fact, the spatial version of MOL demonstrates an interesting effect which is a direct consequence of the use of LLVM. A very short performance comparison is given here to illustrate this.

Although one might normally expect that code which has been automatically generated would be slower than hand written code due to the abstraction penalty, in this case it is found that the generated code is in fact almost an order of magnitude faster. The large difference in performance shown in Figure 5.2 is made less surprising by trivial inspection of the generated LLVM IR code, which yielded use of vectorised instructions, due to LLVM's optimisers (in this case, for the AVX and SSE enabled Intel processor). All vectors in the spatial MOL language are stored as Terra `vector` types, which are consequently mapped to AVX and SSE structures. This allows LLVM to make straightforward code optimisations. The MOL library stores various functionality for linear computations using vectors by providing abstracted Terra operators specified in meta-methods. Terra automatically applies these by inspecting metatables.

Listing 5.12 gives a very simple example of the spatial version of the MOL language. A similar structure can be seen to the Lattice-based MOL language. More examples of both are found in the next section.

The code shown in Listing 5.8 was compared in terms of raw performance against the C++ code in Listing 5.7. The results of this were given in Figure 5.2.

Another advantage of MSP in this instance is obtained. Automatic parallelism is an important concept, especially since clock speeds are no longer increasing, but the number of processors and cores are [221, 161]. Unfortunately

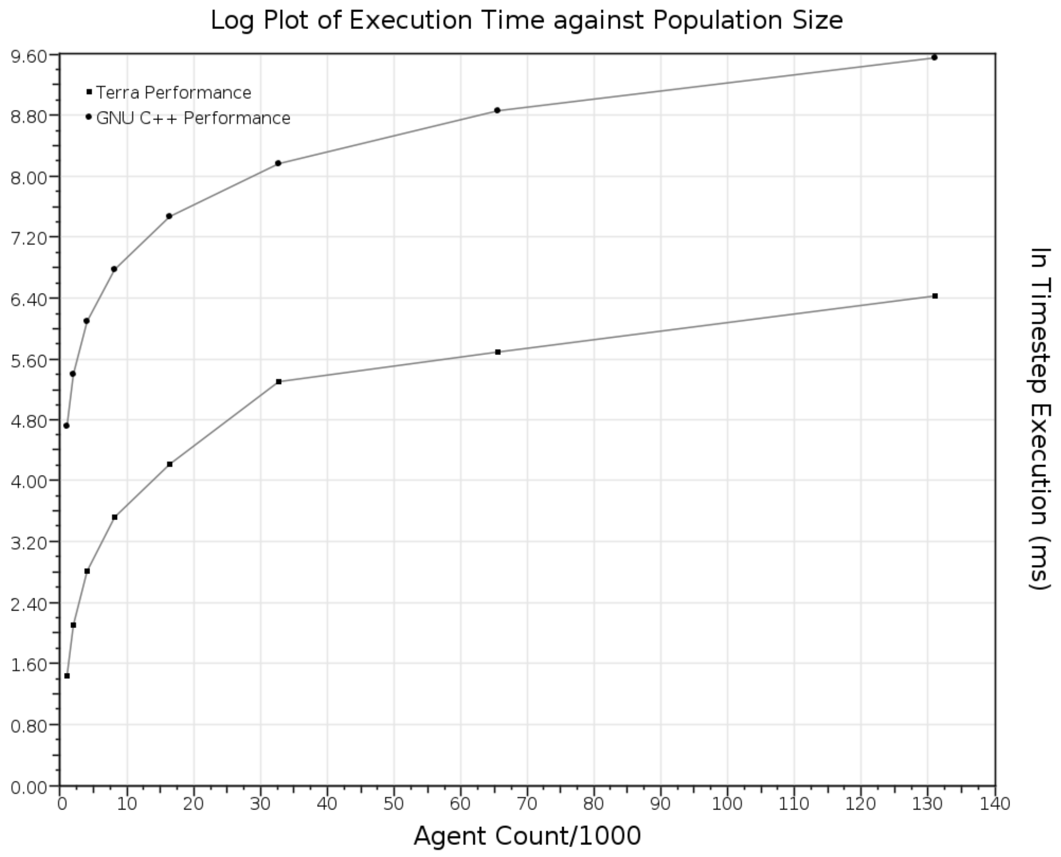


FIGURE 5.2: Performance comparison between hand written GNU C++ code (Listing 5.7) and Terra (LLVM) compiled code (Listing 5.8).

```
float3 goal = make_float3( 0.1f,0.2f,0.3f );
for (int i=0; i < params->agent_count; ++i) {
  float3 *mypos = (float3*)(h_positions + i*3);
  float3 *myvel = (float3*)(h_velocities + i*3);

  *myvel = *myvel + (goal - *mypos) * 0.001;
  *mypos = *mypos + *myvel * 0.1;
}
```

LISTING 5.7: C++ code used for comparison against the MOL code in Listing 5.8. The code uses some typedefs from CUDA.

```
mol
  defvar goal = {0.1,0.2,0.3}
  move towards goal by 0.001
  pos = pos + 0.1*vel
end
```

LISTING 5.8: Custom DSL code used to compare against the GNU C++ code in Listing 5.7.

parallel computing is not always trivially easy to accomplish. It would be interesting to explore the role that recent paradigms such as MSP play in automatic parallel code generation. It is known that expertise can be embedded as function generators using MSP, and it appears that parallelism can also be. Also interesting to note is that the MOL compiler is itself unaware that there is automatic vectorisation of the vector structures. It is very encouraging to observe automatically generated code which performs well in terms of objective and wall-clock performance. This is demonstrated in Chapter 6, where different parallelisation strategies automatically generate suitable code for GPUs.

In the next section, some examples of MOL syntax are given. This will serve to showcase the major features of this language.

5.4 Examples and Selected Results

Two examples of the lattice-oriented MOL language are given in this section, and also a more brief example of the spatial version. First, Conway's Game of Life [75] is given in MOL and Lua-Terra in its entirety, followed by a variant of the Predator-prey model [104, 257]. Then, a flocking model reminiscent of Boids [240] is given and discussed.

5.4.1 Conway's Game of Life

($\mathcal{I}$, p. 15)

A sample program written in MOL for Conway's Game of Life, including some extensions written in Lua-Terra is shown in Listing 5.9. Lines 67–82 contain the special DSL syntax. Some additional code is provided as Lua-Terra *extensions*, which make it possible to extend code using the lexical environment at parse time. Much of the functionality presented as extensions could easily be moved into a separate library for future use. In the spirit of the advantages that MSP afford, this also allows an opportunity to write self-specialising code for future models. The code as shown is placed in a Terra script file (extension `.t`). Lines 1–67 in fact contain valid Lua code, where any constructs beginning with the keyword terra are first converted to first class Lua values.

Extensions to MOL are written in the form of a Lua function which constructs a Terra expression. This is later spliced into the Terra code generated by the MOL compiler. The MOL compiler passes the lexical environment with all its symbols by reference through the argument env. Examples of these are on lines 3, 19 and 38. It is also possible to write Terra functions and use these directly in MOL code, operating on variables and globals in scope. The macro-style MOL extensions take the form of a function, which accesses local and global variables by name. The env Lua table contains all parsed symbols, including ones that are defined by the compiler: wolattice, rolattice , me, agent_count, score, posx, posy, posz, up, down, left , right, back, front, neighbour, nx, ny and nz.

Currently, MOL supports the bare minimum of program control constructs. As seen in Listing 5.9, if-then constructs are supported, along with local variables and constant global variables.

The frame compute event function discussed earlier was used to gather data on the Game of Life simulation in Listing 5.9. Some metrics are computed over this simulation, including live/dead fractions as well as like-like (LL) bond fractions (shown in Figure 5.3). In this case, there are only two cell states (unlike Q-state generalisations of the Game of Life [94]), therefore it can be expected that LL-bond fractions would loosely correlate with the cell live/dead fractions. Nevertheless these are still provided to serve as demonstration. As can be seen, the model reaches equilibrium around time step 275.

Quantitative measures are not to be ignored during the development of a modelling platform. Packages such as NetLogo [279] demonstrate this to some extent, using real-time graphing facilities to aid understanding of various

```
-- not shown: reset function.

function birth(env)
 local wolattice
 local posx = env:localenv()[”posx”]
 local posy = env:localenv()[”posy”]
 local posz = env:localenv()[”posz”]
 wolattice = env:localenv()[”wolattice”]

 local lbirth =
  terra(wolattice: &int, x: int, y:int, z:int)
   wolattice[
    [library.scalar_index_periodic(x,y,z)]
   ] = 1
  end
 return ‘lbirth(wolattice, posx,posy,posz)
end

function death(env)
 local wolattice
 local posx = env:localenv()[”posx”]
 local posy = env:localenv()[”posy”]
 local posz = env:localenv()[”posz”]

 wolattice = env:localenv()[”wolattice”]

 local ldeath =
  terra(wolattice: &int,x: int,y: int,z: int)
   wolattice[
    [library.scalar_index_periodic(x,y,z)]
   ] = 0
  end
 return ‘ldeath(wolattice, posx,posy,posz)
end

-- function to compute number of
-- adjacent live cells
function get_neighbour_count(env)
 local rolattice
 local posx = env:localenv()[”posx”]
 local posy = env:localenv()[”posy”]
 local posz = env:localenv()[”posz”]
 rolattice = env:localenv()[”rolattice”]

 -- opportunity to specialise for
 -- 2D or 3D here without effort
 local getcount =
  terra( rolattice : &int, x: int, y:int, z:int): int
   var c: int = 0
   for a=-1,2 do
    for b=-1,2 do
     if not (a == 0 and b == 0) then
      if rolattice [
       [library.scalar_index_periodic(‘x+a,‘y+b,‘z)]
        ] ~= 0 then
       c = c + 1
      end
     end
    end
   end
   return c
  end
 return ‘getcount( rolattice ,posx,posy,posz)
end

-- MOL DSL specific code
agent = mol
 defvar adjacent_live_cells = get_neighbour_count
 if me == 0 then
  if adjacent_live_cells == 3 then
   birth -- this cell will become live
  end
 else
  if adjacent_live_cells >= 4 then
   death -- this cell will die
  else
   if adjacent_live_cells <= 1 then
    death -- this cell will die
   end
  end
 end
end
```

LISTING 5.9: Conway’s Game of Life written in MOL with some Lua-Terra extensions.

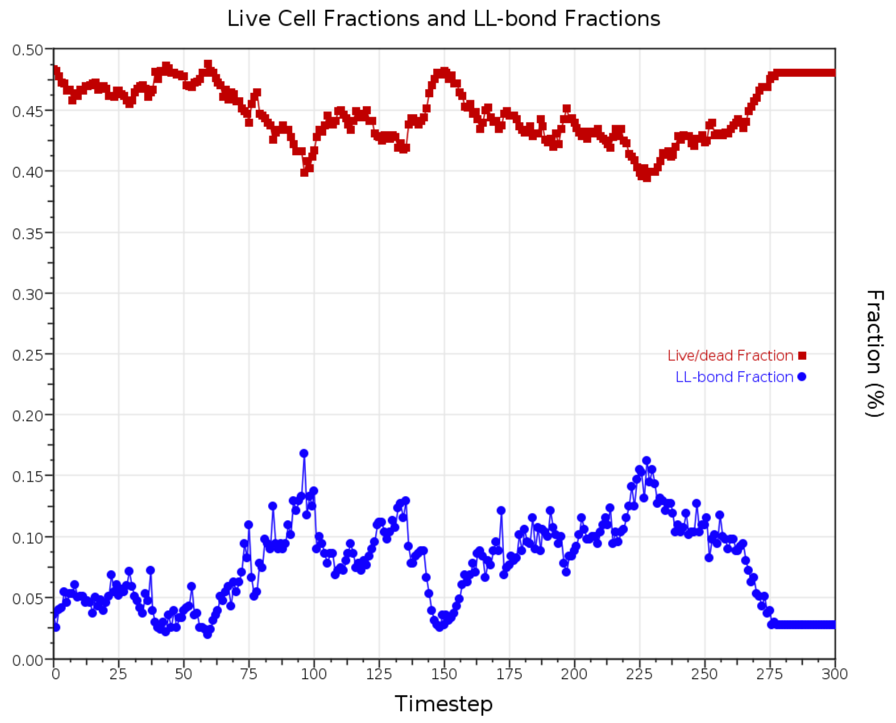


FIGURE 5.3: Live/dead cell count fractions including like-like (LL) bond fractions.

models. While MOL does not support real-time graphing automatically, it can be done with external utilities. In this case, data was collected through the use of a Terra function which uses standard C I/O to write to a file. This file was then plotted.

($\mathcal{I}$, p. 18) 5.4.2 Predator-Prey Model

The Predator-prey model is an agent-based model which posits the problem of predators pursuing prey, and the prey attempting to flee [104, 257]. This model was used to show interesting properties of predator behaviour, including that cooperation among predators by communication is more effective for trapping prey [133]. A very simple variant of this model on a lattice is given in Listing 5.10.

Like the example of the Game of Life in the previous section, this model also uses extensions written as function generators. The computing of the nearest 6 neighbours (NN) or the Moore neighbourhood of 8 cells adjacent to a cell is common in models such as these. This was therefore built into the language instead of requiring an extension to be written for it. This is shown on lines 5–13 in Listing 5.10. In this case, it is used to compute the number of cells of species 1 and 2 (ie. prey and predators). The neighbour keyword is used to refer to the cells independently. This construct was created using the MOL compiler by looping over the adjacent cells (defined by using the neighbours6 or moore8 keywords) emitting the code within the construct once for every cell. The adjacent cells are loaded into the variable denoted by neighbour, and its (x, y, z) position is stored in the variables nx, ny and nz. These special variables are only accessible within such a construct. Another built-in language feature used is the computing of distance. In future, this could be extended to other metric spaces such as Hamming or edit-distances [22].

The code shown in Listing 5.10 generated the results shown in Figure 2.5 in Chapter 2, which is reproduced here for convenience (see Figure 5.4).

```
mol
  defvar count = 1
  defvar pred = 1

  query neighbours6
    if neighbour == 1 then
      count = count + 1
    else
      if neighbour == 2 then
        pred = pred + 1
      end
    end
  done

  defvar predator = 2
  defvar prey = 1

  if me == predator then
    defvar temp = get_closest_prey
    move towards temp

    if pred == 6 then
      die
    end
    split
  end


  if me == prey then
    defvar closepred
        = get_closest_predator ()
    if (distance to (closepred)) < 2
     then
      die
      else
        defvar closeprey
            = get_closest_prey ()
        defvar com = get_predator_com

        -- flee predator (F)
        if (distance to (closepred)) < 3
            then
          move awayfrom closepred
        end
        -- breed (B)
        if (distance to(closeprey)) < 2
            then
          split
        end
        -- overcrowding
        if (count > 7) then
          die
        end
          -- move randomly
        move random 4
        -- seek mate (M)
        if (distance to(closeprey)) >= 2
            then
          move towards closeprey
        end
      end

    end

  end
```

LISTING 5.10: A program written for a variant of the Predator-prey model in MOL.

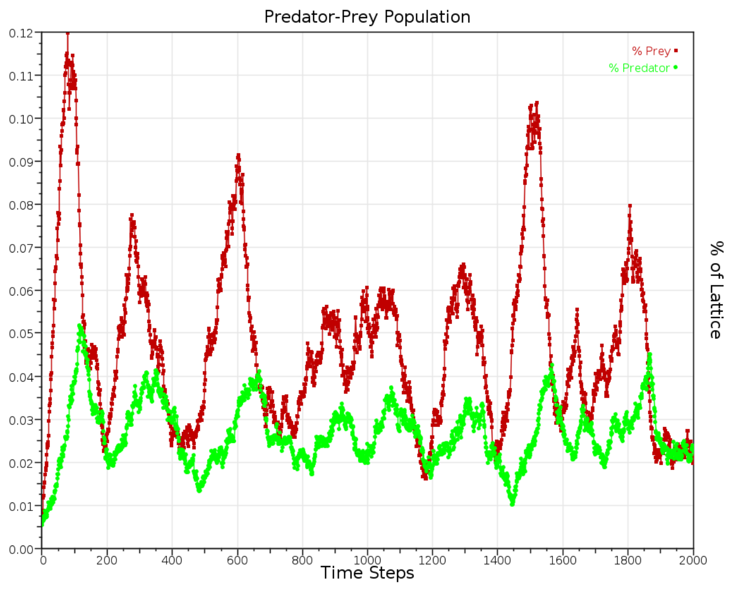


FIGURE 5.4: A plot of the predator and prey lattice coverage (discussed in Chapter 2), duplicated here for convenience).

Performance

It is reasonable to doubt the performance characteristics of the system due to its high level of abstraction. Fortunately, the ability of Terra to generate machine code using the MSP methodology and LLVM ensures that high performing code is generated. As an example, consider version 1 of the reference model which Lytinen and Railsback used [173]. It is an informal and highly simplified model to indicate certain key concepts of agent-based models[7]. Version 1 simply deals with diffusive agents on a lattice. This model was implemented using MOL, and the code for this is provided in Listing 5.11.

```
 1  function diffuse(env)
 2      local posx,posy,posz = get_pos(env)
 3      local rolattice , wolattice = get_lattices (env)
 4  
 5      makeran = ‘std.randint(8)  −− library function
 6  
 7      local mdiffuse = terra(wolattice: &int, lattice : &int, x: int , y: int , z: int ): int
 8          for i=0,100 do −− 100 attempts
 9  
10              var a = makeran − 4
11              var b = makeran − 4
12              var c = 0
13              var newindex = [library . scalar_index_periodic (‘x+a,‘y+b,‘z+c)]
14  
15              if lattice [newindex] == 0 then
16                  var oldindex = [ library . scalar_index_periodic (x,y,z) ]
17                  wolattice [oldindex] = 0
18                  wolattice [newindex] = rolattice [oldindex]
19                  return 0
20              end
21          end
22          return 0
23      end
24      return ‘mdiffuse(newlattice, lattice ,posx,posy,posz)
25  end
26  
27  agent = mol
28      if me == 1 then
29          diffuse
30      end
31  end
```

LISTING 5.11: Implementation in MOL of the first reference model used by Lytinen and Railsback [173]. The lattice reset function is not shown.

As before, the special DSL code is located towards the end of the Terra script file (lines 27–31). Features such as the diffuse function could be easily implemented within the compiler, and just as easily placed within a separate library as well. In this case, the DSL syntax is very minimal.

Though in testing only one CPU was used in the multi-core Intel i7 testing machine, this is a more powerful machine than the i3 CPU used by Lytinen and Railsback [173]. Therefore, the comparison here should be taken as being purely indicative and approximate. With display enabled, the MOL code in Listing 5.11 completed 1000 frames of the first testing model in 1.87 sec which was the total runtime, excluding setup[8]. This is less than the 6.3

[7]Lytinen and Railsback use this to compare NetLogo with ReLogo.

[8]This was averaged over five separate run times.

sec reported by Lytinen and Railsback for Netlogo (Java-based), however, they note that the peculiarities of their testing models meant that performance results obtained from them is also only indicative. The simulation contained 100 agents, which is the same size Lytinen and Railsback used.

A somewhat larger divide in performance was observed in display-less execution. Lytinen and Railsback note that performance increased from $3 - 20\%$ from the 6.3sec reported when rendering was disabled. In the case of MOL, the speed-up achieved with no display was 91%. Ignoring all memory copies and other overheads, the raw computing time spent by the compiled MOL code in computing a single time step is approximately 11.5 msec.

A more thorough example of the usage of MOL and its applicability in ABM can be found in Chapter 7, where a carefully designed algal photobioreactor was implemented.

5.4.3 Flocking

($\mathcal{I}$, p. 16)

Presented here is a flocking model reminiscent of the Boids model [240]. It showcases the spatial version of the MOL language. An example of what this model could look like when executed is shown in Figure 5.5. Similar constructs are used in this version of the language (see Listing 5.12), including the query construct. The main difference is that the concept of neighbourhood is replaced with a sphere which has a restrictive radius. Each agent has a spherical neighbourhood. Arguably, this is more compute intensive than lattice-based simulations because spatial locality in memory is not always guaranteed.

```
agent = mol
        defvar count = 0
        defvar av_vel = {0,0,0}
        query neighbours
                count = count + 1
                av_vel = av_vel + nvel
        done
        defvar closest = select closest in group

        if (distance to closest) < 4 then
                move awayfrom closest
        end
        move towards groupcentre
        move towards {0,0,0}
        vel = vel + 0.5*av_vel

        vel = 0.9*vel
        pos = pos + vel
end
```

LISTING 5.12: Spatial MOL code for a simple Boids-like model. Here, continuous space is assumed, instead of a discrete lattice.

So far, only a portion of the ABM process has been discussed in terms of MOL. What remains is to calibrate and validate models that are written in MOL. This is typically a tedious and time-consuming process. It is uncommon to have a lack of ambiguity in parameter settings in systems like these. A good example is that of the Particle Swarm Optimiser (PSO) discussed in Chapter 3. Depending on the two parameters given by the operator, the global optimum of a function can be found consistently, inconsistently, or never. Unfortunately, the majority of parameter sets tend to provide inconsistent results. A similar problem which is related to this is that of model *structure*. Sometimes there are ambiguities in model design, and one is left to experiment empirically. This problem is discussed in detail in the next chapter.

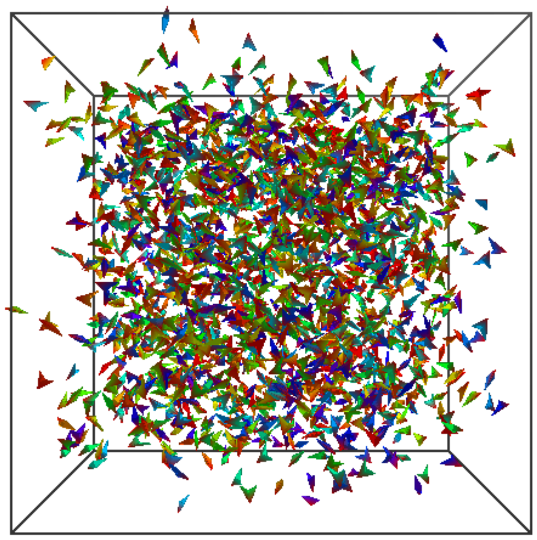


FIGURE 5.5: A screenshot of a flocking model implemented in MOL, using code similar to that of Listing 5.12.

5.5 Discussion

A new domain-specific language named "MOL" has been introduced. It makes use of the recent development of multi-stage programming (in tandem with LLVM) in order to compile programs written in a simple, extensible syntax to machine code at runtime. The use of LLVM ensures that up-to-date code optimisations are used without any additional effort.

Due to the platform-independence of LLVM IR and the higher level Terra code, it is possible to extend the compiler simply by wrapping the code generator with external libraries such as multi-threading, MPI or CUDA. MOL was designed from the beginning to be capable of extension in this direction with relative ease, and this is what the next chapter concerns itself with.

There are some additional advantages to this language which supplement the requirements of the next chapter. It is a very simple language, which usefully shares its lexical environment with Lua. Models can be described easily and with a clean syntax, and all the power of a general purpose language is subsumed by Terra and Lua. Extensions can be written using Terra, and can also be written using C/C++ code, which is a direct consequence of the use of Clang from within Terra.

Extensions can also be made modular, and reused in different models. Compiled external libraries can also be used from within the same Lua-Terra file containing the model. It is also possible to use another domain-specific language in the same model script file. These advantages are obtained simply by the use of Terra.

This language does not, however, provide simple conversion between continuous and discrete lattice spaces. While Terra supports some debugging functionality, this has not been integrated into this language either, making it difficult to debug, especially when an error is made within an extension.

A greater problem is that of extension writing. It is anticipated that when this language is used by the public, it would have had sufficient development effort to make a substantial library of common functions. However, it is not unreasonable to assume that users may require an extension for certain functionality from time to time. This requires being able to write Terra code, which is a low-level language. It is certainly possible to forego the special syntax of the MOL language and simply write the model entirely within Terra. However, as will be detailed in the next chapter, this special syntax is indeed very useful for providing pre-processor–like functionality for optimisation.

CHAPTER 6 PARALLEL DOMAIN-SPECIFIC OPTIMISATION IN ABM

Optimisation in the context of Agent-based Modelling (ABM) has been previously reported in the literature, particularly efforts in parameter tuning. Though calibration of parameters is extremely important, the behaviour of a model depends perhaps more on the processes which use these parameters. Improvements in machine learning and combinatorial optimisers have cast new light upon the concept of model induction and structure optimisation. Yet, performance issues plague these algorithms.

This chapter presents critical modifications to MOL, the ABM language proposed in Chapter 5, in order to accomplish structural optimisation of agent-based models in a pragmatic manner. Performance improvements are paramount due to the complexity of this optimisation task. Further extensions of MOL in terms of data-parallelism are presented in this chapter in order to reduce difficulties brought on by performance issues.

This chapter and Chapter 5 extend upon work previously published by the author the proceedings of *The 13th IASTED Int. Conf. on Artificial Intelligence and Applications*[1], the proceedings of the *INMS Postgraduate Conference, Massey University* in 2013[2] and an article which was submitted to the *International Journal of Modelling and Simulation*[3].

6.1 Introduction

NOT ONLY IS ABM CONSIDERED A GENERATIVE APPROACH to macro-level phenomena [59], the process of ABM has itself become the subject of top-down approaches. Epstein earmarked Evolutionary Algorithms (EAs) in 1999 for optimising the processes within agent-based models by noting that current research examining the parameter space of an agent-based model systematically is far easier than examining the immense *behaviour space* (in terms of the possible rules) [59]. Literature dealing with parameter calibration for ABM essentially reinterpret the hand calibration process as an optimisation problem [29, 28, 271]. Similarly, the choice of precisely how a model is formulated in terms of rules, processes and structure can also be reinterpreted as an optimisation problem, but perhaps a more difficult one.

There have been several attempts in the past towards applying automatic optimisation techniques to ABM to calibrate parameters. Calvez and Hutzler combined genetic algorithms and ABM [28], as did Said et al. [250], both

[1]A. V. Husselmann and K. A. Hawick. Multi-stage high performance, self-optimising domain-specific language for spatial agent-based models. In *The 13th IASTED International Conference on Artificial Intelligence and Applications*, Innsbruck, Austria, February 2014. IASTED

[2]A. V. Husselmann and K. A. Hawick. Towards high performance multi-stage programming for generative agent-based modelling. In *INMS Postgraduate Conference, Massey University*, October 2013

[3]A. V. Husselmann, K. Hawick, and C. Scogings. Model structure optimisation in lattice-oriented agent-based models. Technical Report CSTN-222, Computer Science, Massey University, 2014. Submitted to the International Journal of Modelling and Simulation (ACTA Press)

of which focus on calibrating the parameter sets of the models. Stonedahl and Wilensky combined the popular ABM platform NetLogo with a parameter calibration algorithm [271]. In these systems, there is only provision for calibrating scalar parameters. While adequate in some cases, such optimisers are invariably limited in that they cannot extensively modify the semantic structure of the model on which they are built. Instead, suboptimal structure can render the meaning of a well-calibrated parameter useless.

There have been numerous attempts to induce models and to optimise the *structure* of agent-based models in the past. Examples include the recent work of van Berkel [283, 284] in 2012, which involved the Grammatical Evolution [249] algorithm to generate NetLogo [279] programs from a set of predetermined "building blocks". While van Berkel concedes that determining a good fitness function is a difficult challenge, not much emphasis is given on the choice of building blocks, other than being user-provided, and that some can be reused from other distributed algorithms. Both these issues are portrayed as open problems in Genetic Programming (GP) by O'Neill and colleagues [205]. One may argue that carefully articulating the set of components from which to induce a model is in itself a problem with comparable difficulty to model construction [205]. It is difficult to choose the set of "building blocks" necessary to limit the search space sufficiently and to ensure the optimum can still be found.

As a further example, Junges and Klügl in 2010 evaluated learning classifier systems, Q-learning and Neural Networks for behaviour generation [134]. No clear winner among the algorithms was found by the authors. Later in 2011, the authors investigated Genetic Programming in the same context [135], integrating it with SeSAm, a Java-based platform [147], but did not compare directly with reinforcement learning or other techniques. They did note in 2012 that Reinforcement Learning and GP proved to be more useful for their ability to generate human-readable results [136].

The most recent work of Junges and Klügl focus on a specialised modelling methodology adapted for machine learning [137] termed MABLe (Modelling Agent Behaviour by Learning). The optimisation algorithm is abstracted as a "learning core", and may be any one of Q-learning, the XCS learning classifier system, or Genetic Programming. This perhaps further signifies that none of these is clearly better than the others in all cases. This finding is further supported by the "No Free Lunch" theorems of Wolpert and Macready [294], previously discussed in Chapter 3.

A single overarching objective function is among the steps of the Junges and Klügl MABLe methodology. Objective functions are notoriously difficult to craft, and its successful derivation is often in fact a considerable portion of research, such as in softbot team coordination [170] (a multi-agent example).

The work of Privošnik in 2002 resulted in an evolutionary optimiser which evolved agents operated by finite state machines (FSMs) to solve the Ant Hill problem [231]. Like Jim and Giles, Privošnik et al. observed an increase in fitness with an increase in the number of states in the FSMs. The potency of representing an agent in terms of FSMs can be demonstrated by the formalism of the X-Machine specifically for ABM [37]. However, like other techniques, a well-articulated set of observations (percepts) and actions is still predetermined.

A considerably limiting factor in these experiments is that of performance. Quite often, these algorithms are so intensely compute-bound that scaling to a reasonable system size, achieving a desired timestep, or even reaching a desired outcome is impractical. As exemplified by meta-optimisation discussed in Section 3.5, evaluating the fitness of a candidate model involves executing it and gathering some quantity; in the case of stochasticity, several times for an average. Evaluating a time step of a population-based model optimiser involves computing the fitness of n models in a population. This expense quickly becomes unmanageable for larger n. This motivated Privošnik to propose a heuristic method in fitness evaluation to reduce computation expense [230]. Trajectory-based methods (single candidate successively improved) such as Simulated Annealing [145] are perhaps more suitable for optimising models with regards to performance, but they do not benefit from other concurrent explorations and communications elsewhere in the search space.

Though the Java language has advanced considerably in recent years, it is still at least 1.09 times slower than C++ for computing benchmarks such as the Delaunay triangulation algorithm [78]. As shown in Chapter 5, there is also some performance increases obtainable from custom written software as opposed to Java-based ABM platforms. With these drawbacks in mind, the integration to the modelling platform SeSAm simplifies the process greatly for the user. However, like Netlogo [279] and RePast [38], SeSAm [147] is also Java-based. It appears that the larger and more mature ABM platforms are Java-based, which perhaps further alludes to the demand for ease of use from users.

Though there are several algorithms which are capable of calibrating structure in an agent-based model, they do not always produce human-readable results, and this is counterproductive for ease of use. It is common knowledge that algorithms such as Neural Networks are more difficult to interpret once calibrated, especially when compared against code constructed from the recombination of human-written code. Junges and Klügl, however, demonstrate how rules can be extracted from Neural Networks [134] by providing inputs and tracing activation through the network to obtain an input/output pair, which can be essentially interpreted as a rule. An advantage of Genetic Programming is that its outputs are inherently human-readable, a property shared by other similar evolutionary algorithms, and does not require post-processing.

The problems described above are all addressed in this chapter by making modifications to a Domain-Specific Language (DSL) described in the previous chapter. Optimisation is approached by using work from previous chapters on EAs for combinatorial optimisation. Performance is then also addressed by using data-parallel techniques used in earlier chapters. Figure 6.1 shows a visualisation of the built-in optimiser developed in this chapter searching through suitable rule sets for the Predator-Prey model. In this case, the optimiser was searching for a ruleset for prey agents which minimises the *number* of prey agents in the model. This is only an illustrative example, but might even give some insight into the question “which prey behaviours cause predators to be more effective?”, which is a question that ecologists might want an answer to or insight into.

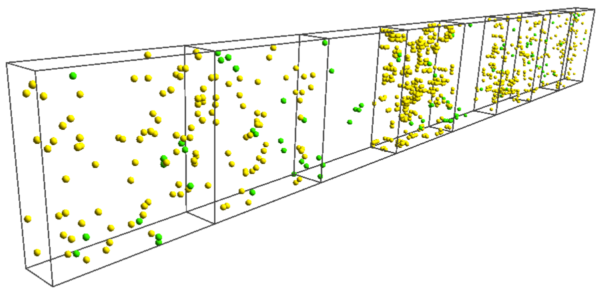


FIGURE 6.1: A visualisation of a population of 8 candidate models being evaluated. The optimiser is attempting to find a ruleset for the prey which minimises the *number* of prey in the model, an illustrative example. The program for this is shown in Listing 6.2.

In Chapter 5 a DSL named MOL was proposed to improve the modelling experience with extensibility via Multi-stage Programming (MSP), and added performance due to being machine-code compiled. However, MOL was not necessarily enhanced beyond its ability to represent models. This DSL can be extended to assist in the search for a suitable model structure or design, and do so while harnessing the parallel computing power of Graphical Processing Units (GPUs). This is approached in this chapter.

To give further rationale on the choice to extend the MOL language for this purpose, it is hypothesised this language will:

1. Enable model induction with a simple and easily modifiable/extensible language
2. Allow selective fine- and coarse-grained model induction, allowing users the ability to optimise any aspect of their model in variable granularity
3. Allow the user to define internal building blocks of the optimiser algorithm directly using domain-specific knowledge
4. Avoid run-time code interpretation and abstraction penalties by using MSP
5. Take advantage of the inherent reduction in search space, and also encourage the operator to expose only what is absolutely necessary to optimisation
6. Avoid manual re-compiling overhead during the modelling process

As in Chapter 5, focus in this chapter is given to lattice-oriented models, but spatial models are also briefly discussed. The rest of this chapter deals with first extending the MOL compiler architecture to include an optimisation phase in Section 6.2, and then the language is extended for data-parallelism using GPU hardware in Section 6.5. Following these sections, some performance and convergence results are presented for selected problems in Section 6.6.

6.2 Model Structure Optimisation using MOL

In the methodology described here, instead of a modeller providing fine-grained and carefully articulated local behaviours, the modeller instead provides an objective function and some possible behaviour within simulation code. The difficulty in both these tasks are comparable: small variations in local behaviours as well as different choices of objective function can both lead to radically different results. However, the novelty in this approach to the problem lies in the elegance with which a model can be prototyped with all aspects including optimisation encapsulated within it, without suffering performance penalties. Performance received considerable attention in this language. Also, in cases where an objective function is more natural to use, it can be a great advantage to have the behaviour generated from it, or at least provide a suggestion which can be used as the basis for manual extension. By allowing a modeller to express most of a simulation with reasonable certainty, and other parts with annotated uncertainty and clear objectives, an underlying optimiser can be useful in assistance, as will be demonstrated later in this chapter, and also Chapter 7.

Apart from choosing a suitable optimiser, a number of other issues must also be carefully considered in order to successfully augment MOL for structural optimisation. The issues are (briefly):

1. Clear unambiguous syntax
2. Straightforward selection of optimisation constraints and strategies
3. Computing of objective functions
4. Limiting the search space as much as possible without compromising efficacy

5. Straightforward selection of optimiser parameters
6. Effective elimination of stochastic variation in fitness
7. Improving prohibitive wall-clock performance
8. Appropriate visualisation and performance indications for user feedback
9. Careful implementation of primitive memory facilities

One of the factors which sets this new language apart from other model induction attempts is that the syntax used for optimisation is itself a constraint upon the search space. Techniques such as Genetic Programming have been used in the past for evolving entire models [284, 135] which were provided with user-defined actions and perceptions of agents. In the case of MOL, only relevant actions and perceptions are provided, but they are given in terms of a *potential solution* given by the user. This kind of syntax serves to (1) provide a good starting solution and more importantly (2) limit the immense search space. It is reasonable to assume that the dynamics of a system is at least partially known by the expert, and therefore, it is worth optimising only the *uncertain* portions. It is certainly noteworthy that this kind of optimisation can also be done if a model was constructed and a suitable optimiser was used for a portion of that model, but that requires model-specific optimisation. MOL differs in the sense that the model is given in the same language, and optimisation requires only pre-processor style directives, completely eliminating the need to write custom code for optimisation.

A method is needed to (1) express an objective function and (2) define when it is measured. At present, the compiler adds a statement to the code of an agent which computes and adds the fitness to a running total for the agent. For example, the MOL code shown in Listing 6.1 provided as part of an agent's code will provide the variable score with a value of zero for all but time step 1000. On every frame, the agent's code is executed, and the variable score will be added to a running total in an array maintained by the host C++ program. The if statement serves to define *when* the number of live cells will be measured (count_live_cells is a macro in this case). This is assuming that one evaluation of the program is exactly 1000 time steps. The syntax for defining score as the fitness will be given later. Of course, the code shown in Listing 6.1 is arbitrary. This can be replaced by any quantity and any measurement method desired, even including agent memory, discussed later.

```
defvar score = 0
 if timestep == 1000 then
    score = count_live_cells
end
```

LISTING 6.1: User-defined MOL code to compute a quantity representing the score of an agent.

Given a method for expressing the objective function and for measuring it, some method of configuring the output desired from the optimiser is required. Certain constraints apart from search space define, in large part, the operation of the optimiser, or even what kind of optimiser is used. Three such configurations are implemented by specifying one of the keywords recombination, single, or permutation. These qualifiers tell the compiler which kind of optimisation is required. What is referred to by these qualifiers is some kind of reorganisation of *statements* marked for optimisation. While single and permutation refer to the selection of a single statement only, or a permutation of statements (with replacement), recombination refers to a *disassembly* of the provided code into terminals and nonterminals for recombination. These qualifiers are detailed in Sections 6.4.2 and 6.4.3. The single qualifier is omitted, as it is less useful and applicable than the other two qualifiers.

A complete example of this is given in Listing 6.2. This example is similar to that described in Chapter 5, given in Listing 5.10. In Listing 6.2, lines 41–63 contain what will be referred to as an “uncertain” construct. This term is purely related to the structure of the model, and not of code semantics. The quantity is count_sheep() and the objective is to minimise this (hence the minimise keyword given on line 41).

```
 1 mol
 2  defvar count = 1
 3  defvar pred = 1
 4
 5  query neighbours6 --(NN)
 6    if  neighbour == 1 then
 7     count = count + 1
 8   else
 9     if  neighbour == 2 then
10      pred = pred + 1
11     end
12   end
13  done
14
15  defvar predator = 2
16  defvar prey = 1
17
18   if  me == predator then
19   defvar temp = get_closest_prey
20   move towards temp
21
22    if  pred == 6 then
23     die
24   end
25    split
26  end
27
28  defvar eq = count_sheep()
29
30   if  me == prey then
31   defvar closepred
32              = get_closest_predator ()
33    if  (distance to  (closepred)) < 2
34         then
35     die
36   else
37    defvar closeprey
38                  = get_closest_prey ()
39
40
41    select  permutation to minimise(eq)
42     -- flee predator (F)
43     if  (distance to  (closepred)) < 3
44                  then
45      move awayfrom closepred
46     end
47     -- breed (B)
48     if  (distance to(closeprey)) < 2
49                  then
50      split
51     end
52     -- overcrowding
53     if  (count > 7) then
54      die
55     end
56               -- move randomly
57     move random 4
58     -- seek mate (M)
59     if  (distance to(closeprey)) >= 2
60                  then
61      move towards closeprey
62     end
63    end
64   end
65  end
66 end
```

LISTING 6.2: A program written in the MOL DSL for the Predator-Prey model, containing an uncertain construct (lines 42–64).

As mentioned in Section 1.4 and Section 3.5, the selection of parameters for an optimiser is critically important, and can also be reinterpreted as an optimisation problem itself. Similarly, it is important to choose suitable parameters. The choice of parameters are discussed in Section 6.3, which are, at this point, manually chosen. It is anticipated that this can be extended using concepts from meta-optimisation (see Section 3.5 in Chapter 3).

In cases where the fitness of a model is non-deterministic, it is important to obtain statistically significant scores. This is typically done by averaging multiple evaluations of the objective function. The performance implications of this is addressed in Section 6.3 by the use of data-parallelism on graphics hardware using Compute Unified Device Architecture (CUDA). As with other parameters such as population size required by the optimiser, the number of evaluations to average over is also important. This is chosen by the user (and noted in experiments in this chapter), as a more thorough understanding of the precise variability in the model is necessary to make a suitable choice.

As mentioned earlier, there is considerable computing time involved in this kind of optimisation. It is for this reason that improved performance is sought from data-parallel architectures such as CUDA. While other parallel architectures would also be suitable, graphics hardware was chosen for their high theoretical performance at a reasonable cost. Though these devices are less expensive than grid computers, effort is required to ensure that a program maximises their computing power.

It is also important to consider providing feedback to the user. Visualisation of the population of candidates would involve rendering several different models in real time. Doing so requires a fast rendering method. In this case, cells are rendered using a fast pixel shader with OpenGL. No other optimisations are used. It is noteworthy that this visualisation pertains to the *evaluation* of a population of candidates, and not the syntactic structure of them. Candidate populations have been discussed in terms of visualisation in Section 4.8.

In Section 6.5, the MOL language is modified to generate CUDA PTX instructions through Terra and LLVM. Allowing the MOL compiler to generate CUDA code through Terra and LLVM places some restrictions on data, since local variables in MOL programs only remain in scope for the length of a single execution. Alongside the lattices, a suitable array for storing this data must therefore also be stored on the host. The only static variable throughout all executions (until the end of the simulation) is the lattice site state, which is a 32 bit integer. It is anticipated that future work could extend this to dynamically allocate a host structure array at runtime, and the corresponding GPU memory also. This will allow individual lattice sites to declare static variables (as well as arbitrary state variables) and store data that will remain in scope until the end of the simulation. Instead, static variables are currently stored and accessed within this site state using the backwards compatibility of Terra with C. Terra uses Clang (a C/C++ frontend for LLVM) to compile inline C code within Terra script files. Special C code is written within a model script file to store and retrieve data using bitmasks on the integer cell state variable stored in the lattices.

The next few sections describe various examples of MOL in the context of model structure optimisation. Examples of optimiser configurations are given and discussed. Convergence results of experiments are given in Section 6.4. Performance results are then provided in Section 6.6, following the CUDA-specific optimisations of MOL in Section 6.5.

6.3 The Single-threaded Evolutionary Optimiser

There are many choices in optimisation algorithms, ranging from sophisticated multi-operator evolutionary algorithms [64] to hill-climbing searches and the original Genetic Programming algorithm [149]. The optimiser chosen for recombination is a single-threaded LuaJIT implementation of the algorithm presented in Section 4.4, Chapter 4, and a much simpler evolutionary algorithm also written in Lua is used for single and permutation, as these configurations do not require tree representations.

The software architecture involved in the DSL involves the usual compiler backend and frontend: a parser, type checker and a code generator. A separate optimisation phase is added, not in which code is transformed for optimisation purposes, but to allow for an evolutionary optimiser to adjust the typed internal parse-trees. Optimisation is not intended to be a part of the model simulation process. In contrast, it is intended as a *pre-processor*, which sees to it that a totally unambiguous program is constructed once optimisation has ended. Figure 6.2 shows the complete process of the system. It is essentially identical to an evolutionary optimiser, except that candidates are provided by the special parser as modifiable typed syntax trees, which are edited by the genetic

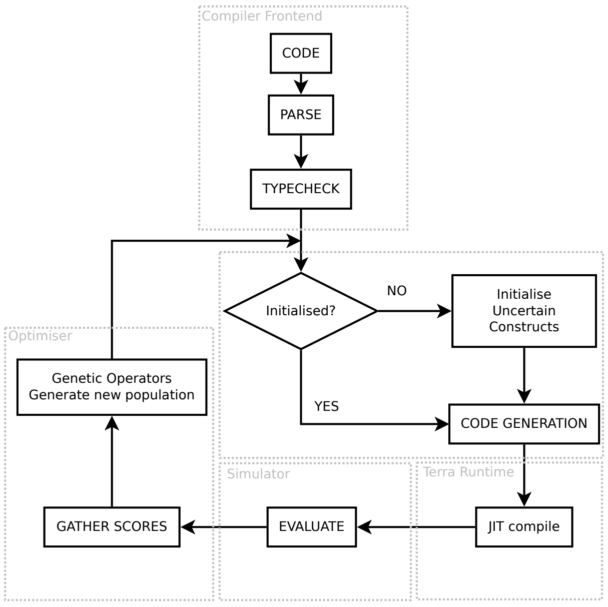


FIGURE 6.2: A flow diagram indicating the underlying optimisation process. It is divided into the use of the compiler frontend, then optimiser initialisation, followed by the runtime environment and fitness computations. Genetic operators are then used to generate a new population of programs by modifying typed trees in memory, and the process repeats.

operators. It should be noted that the compiler and parser were heavily inspired from code samples distributed with Terra [46].

In this process, a user first provides a program with an uncertain code segment within a model. Upon execution of the host program which is written in C++, the Terra runtime parses the provided code immediately, transforming it into a type-checked abstract syntax tree (AST). This tree is what is modified by the optimiser in a later stage. The host Lua script then duplicates this tree several times until a population of N trees is made. At this point, the optimiser searches for an optimisation statement, of which there can be only one within a program. The optimiser then initialises each program's code segment with randomly chosen statements taken from within the provided construct. Depending on the type of construct, there will be one chosen statement (single), or a combination of statements (permutation) with replacement, or a complete recombination of code (recombination).

In summary, the overall process is detailed in Algorithm 13. This algorithm contains all the components generally expected within an EA, and more specifically one which uses genetic operators. The termination criteria shown is simply a maximum number of generations. Computing a new generation of programs is done by the process shown in Algorithm 14.

It is worth noting that backwards-compatibility is maintained with MOL programs which do not contain any of these constructs. This is done simply by disabling the optimiser if it cannot find an optimisation construct within a parse tree. Execution then continues as normal.

6.4 Experiments and Convergence Results

In this section, examples of the different optimisation configurations are given (including single, permutation and recombination), along with their convergence results. Emphasis is given to the recombination construct, since

```
Terra custom parser reads agent description
Read user-provided parameters
Allocate & initialise space for $n$ candidates
{small variations when no uncertainty is present}
Compute and compile a new generation of candidates
Zero all scores
while Termination criteria not met do
    for $x$ frames do
        Execute model programs
        Collect new scores into running totals by model
        Visualise the result
    end for
    Compute a new generation using collected scores
end while
Output best candidate in final population.
```

ALGORITHM 13: The complete simulation process eliminating uncertain constructs.

```
Collect scores
for $x$ candidate models do
    Optimiser performs evolutionary operators
    Pass modified typed tree through code generator
    Wrap generated function code with arguments
    Emit wrapped code
end for
Overall generated code is compiled to machine code
Fn pointer to compiled function passed to C++ via Lua
```

ALGORITHM 14: The process of generating a new generation of candidate models for evaluation.

this is a considerably more sophisticated optimiser than that used for the former two constructs, and requires more detailed explanation.

6.4.1 Single-statement Selection

The trivial case of selecting a single statement among a set of statements is accomplished by using the single keyword. A hypothetical example of this which can be written is given in Listing 6.3. Any unresolved symbols such as become_plasma_cell and count_t_cells are resolved to extensions, written in the style of a macro-like function which generates Terra statements using MSP by the user, and/or stored in a library. Though this requires sufficient programming ability, these can be written by experts and re-used easily. In this case, hypothetically the user might be interested to see which antigen behaviour of die, split, become_plasma_cell and random movement may cause a reduction in the number of T-Cells in the complex human immune response, implemented as a lattice-oriented agent-based model.

```
mol
  if me == ANTIGEN then
    defvar fitness = count_t_cells

     if tcell_attached then
       select single to maximise(fitness)
          become_plasma_cell
           split
          move random 4 --random movement
          die
       end
    end
  end
end
```

LISTING 6.3: An example of single-statement selection in a MOL *uncertain* construct.

6.4.2 Permutation Extraction

Listing 6.2 shows a simple predator-prey model [104, 257] which uses a select statement with the permutation keyword. Statements such as these will be referred to as "permutation statements", similarly for single and recombination. More details of the predator-prey model are given in Chapter 2, specifically Section 2.1.3. Essentially this type of model seeks to mimic the dynamics (and perhaps the reproduction of a specific phenomenon) in a struggle for survival between prey and predator animals.

Another example of the permutation statement is a flocking simulation reminiscent of Boids [241, 240] shown in Listing 6.5. This example uses the spatial version of MOL. The agent-based model is initialised with agents at random locations and with random velocities. Then, through the query statement (reminiscent of the Proto int−hood statement [15]), it is determined how many agents are in the neighbourhood of this agent (a sphere of predetermined size). This program is executed by all agents, therefore, there are intrinsic vel and pos globals giving access to an agent's velocity and position vectors. The closest agent is computed by a statement of the form select closest in group. The objective is arbitrarily defined to attempt the maximisation of the number of agents in the neighbourhood of each agent. The evolutionary optimiser chooses a combination of the statements in the permutation statement with replacement. The algorithm is much simpler than that of a full Genetic Programming implementation because the statements can be considered as terminals.

Should the user require, statements can be grouped so that the optimiser considers it as a single statement in terms of optimisation. This is done by using syntax similar to that shown in Listing 6.4 on lines 6–9.

```
1  mol
2    defvar fitness = count_t_cells
3      select single to maximise(fitness)
4        become_plasma_cell
5         split
6        select all
7           kill_surrounding_cells
8          die
9        end
10       move random 4 --random movement
11       die
12     end
13   end
14 end
```

LISTING 6.4: An example of a select all statement to group statements.

During testing, parallelism was disabled. Fitness values were not averaged, meaning that results from the evaluation stage were subject to error, though this is only for the purpose of a proof of concept. Evolutionary algorithms are capable of handling noise in objective functions, but this is a simple optimisation test. Selecting a combination of the statements given in the select permutation statement is almost as trivial a problem as single-statement selection. However, should there be a large number of these statements, then an optimiser such as this would be more suitable. In this case, a simple example was chosen to illustrate the use of MOL in structural optimisation.

Parameters used for various components of the language and optimiser are shown in Table 6.1.

The optimiser yielded the code shown in Listing 6.6 for the select permutation statement. A screenshot of the model using this code block is shown in Figure 6.4, whereas all the models together during optimisation is shown in Figure 6.3. As the optimisation goal was to maximise the number of agents in the neighbourhood, it makes sense that agents should cluster together and not separate themselves. Of course, if it was necessary to ensure separation between agents, the first statement specified as possible behaviour could be placed before the select permutation statement, thereby making it mandatory.

($\mathcal{I}$, p. 71) 6.4.3 Recombination

To evaluate the recombination statement, the Santa Fe Ant Trail problem was chosen [149]. This problem was previously described in Section 4.1.2. For convenience, the function and terminal sets were

$$\mathbb{F} = \{\text{IfFoodAhead (I)}, \text{ProgN2 (P)}, \text{ProgN3 (Q)}\}$$

```
mol
    defvar m = 0.001
    defvar av_vel = {0.0,0.0,0.0}
    defvar count = 1
    query neighbours
        count = count + 1
    done
    defvar c = compute closest in group
    defvar origin = {0,0,0}

    select permutation to maximise(count)
        if (distance to c) < 3 then
            move awayfrom c
        else
            move towards c
        end
        move towards c
        move towards origin
    end

    vel = 0.995*vel
    pos = pos + vel
end
```

LISTING 6.5: A spatial ABM program written in the custom DSL, with selective uncertainty.

Parameter	Value
Time Steps per Generation	260
Number of Generations	10
Agents per candidate model	64
Number of candidate models	32
Max magnitude of initial velocities	0.16
Max turn velocity	0.1
P(crossover)	0.8
P(mutation)	0.01
Cube dimensions	30x30x30 units

TABLE 6.1: Additional parameters used in the simulation and optimiser.

```
move towards origin
vel = vel + 0.5*av_vel
move towards c
move towards c
```

LISTING 6.6: The optimised select permutation statement.

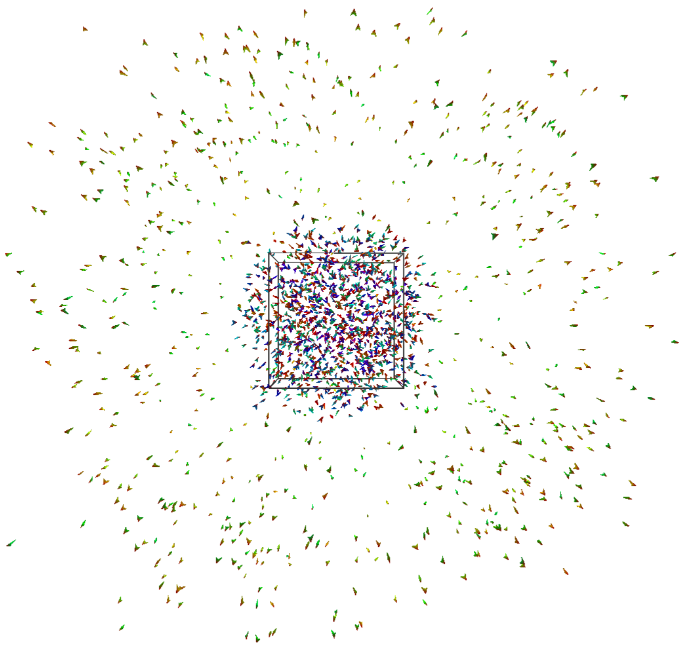


FIGURE 6.3: A screenshot of the optimiser in evaluation stage, gathering fitness scores from 16 separate candidate simulations, using the program specified in Listing 6.5 (with agents superimposed). All candidate models are superimposed. The cube in the centre of the image is where all individual agents are intialised.

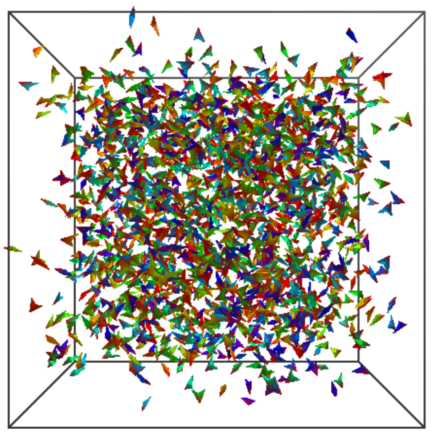


FIGURE 6.4: A screenshot of the optimised program from Listing 6.5 using the code block from Listing 6.6.

and $\mathbb{T} = \{\text{Move Forward (M), Turn Right (R), Turn Left (L)}\}$. It is not as straightforward to form a solution to this problem which achieves full score.

```
if mytype == 1 then
  defvar fitness = 0
   if timestep == 200 then
     fitness = count_food
  end
   select recombination to minimise(fitness)
      if food_ahead == 1 then
          move_and_consume
      else
          turn_left
          turn_right
      end
   end
end
```

LISTING 6.7: The initial code given for the Santa Fe Ant Trail model induction attempt.

The MOL code excerpt provided for this is given in Listing 6.7. The only code not shown is a small piece of code to ensure that only one terminal is processed during execution, this prevents the algorithm from expanding rapidly and simply iterating over the entire lattice.

It is perhaps immediately clear that this initial code would not solve the problem, but it provides all but one of the terminals and nonterminals to construct a working solution. Two other nonterminals are provided separately within the optimiser. These are the ProgN2 nonterminal, and the ProgN3 nonterminal. These two functions simply execute all their arguments sequentially.

Candidates in these populations are represented by using the Karva language of Ferreira [64], and expressions (frequently abbreviated to K-expressions) are often written: `Q-*+/-abca/a-bbadac`. This string encodes a tree, where the symbols `Q`, `+`, `-`, `*`, `/` are 2-arity nonterminals and a,b,c,d are terminal symbols. This tree is a "genotype", since not all of its symbols are included in the final "phenotype" (ie. executable program). It is interpreted by placing the first symbol at the root, and successively adding tree arguments one by one, from left to right, level by level until the tree is complete.[4]

The initial parsing of the code given in Listing 6.7 would occur as normal, but a separate pass constructs the set of nonterminal and terminal symbols from the code given in the select block. The if statement within the block (including its condition) is stored as a 2-arity nonterminal and coded as I0. The I indicates that it is an if statement, and the zero indicates that it is the first if statement encountered. Other if statements with different conditions are stored as I1, I2 and so on. Terminals such as move_and_consume is stored as N0 (anonymous Lua expression), and the other symbols are stored as N1 and N2. For these tests, the statements given in the construct are not used, other than to construct a symbol database. Future work includes decomposing this code in the same fashion and inserting it into the first generation. In the case of this simulation, the following unambiguous symbols will be

[4]For a complete discussion of Karva, see Section 4.2.

used to ease readability: `M–move_and_consume`, `P–ProgN2`, `Q–ProgN3`, `L–turn_left`, `R–turn_right`, `I–IfFoodAhead`. The fitness is computed at timestep 200, using the code just before the select block.

Together, the constructed function and terminal sets are used to construct random k-expressions, enable simple crossover and mutation operators, and construct typed ASTs for inclusion into typed candidate program trees. The most important purpose that conversion into, and out of k-expressions using code trees is to allow crossover and mutation to take place on k-expressions.

In order to implement the terminals turn_left and turn_right, it is necessary to store a direction state variable. This was embedded into the cell state variable (32-bit integer stored in the lattice). Bitmasking functions written in C and used within a MOL extension allowed the agents to retrieve and update these state variables. Storing this kind of information in this manner is not absolutely necessary, but it is done to enable simple compatibility with the parallel code generator at a later stage.

Additional parameters chosen for this model are shown in Table 6.2.

Parameter	Value
Total Generations	10000
Timesteps per Generation	200
Population Count	200
Grid Size	32 by 32
P(mutate)	0.1
P(crossover)	0.8
Program Head Length	17
Total Genotype Length	$17 * 2 + 1$

TABLE 6.2: Parameters used for optimising model structure for the original version of the Santa Fe Ant Trail problem.

Figure 6.6 shows 1000 generations of a sample model optimisation run. In this instance, lower scores indicate less food tokens left on the lattice after execution of 200 time steps. The target is to reach zero. As seen in the plot, mean fitness decreases steadily, and minimum fitness in the populations drop significantly at irregular intervals. Finally, at around generation 950, the best possible fitness is achieved. The candidate found is discussed below.

Computing time was approximately two and half hours, and a program was generated which obtained the best possible score. The program tree is shown in Figure 6.5, and the corresponding k-expression was:

QRPLQMQIPIILLRIIPRLRLRRRRL

Which does not show intron symbols. It is interesting to note that this program needed 26 symbols, whereas the hand-written solution (which was IMPLPIPMRRPLPRPIMML), required 19.

At this point it is wise to consider the problem of performance. The Santa Fe Ant Trail problem is of inherently low complexity, since there is only one active agent on a lattice, which must execute code. In addition, it is completely deterministic, requiring no averaging to obtain a statistically significant score. However, two performance characteristics are considered, the first is the time taken to compile the programs, and the second is to execute them all for one timestep.

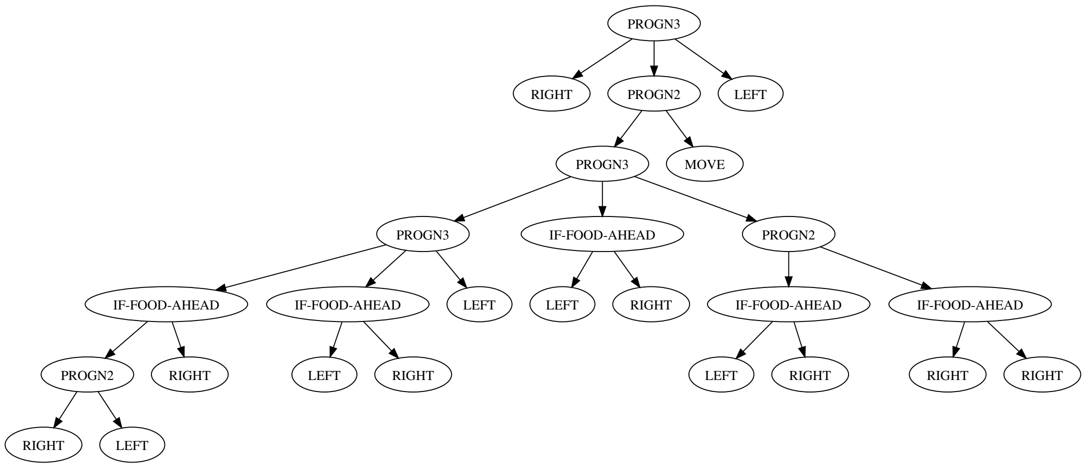


FIGURE 6.5: The best candidate generated by the MOL recombination optimiser (executed using the code shown in Listing 6.7). The score this candidate achieved was zero, which is the best possible score.

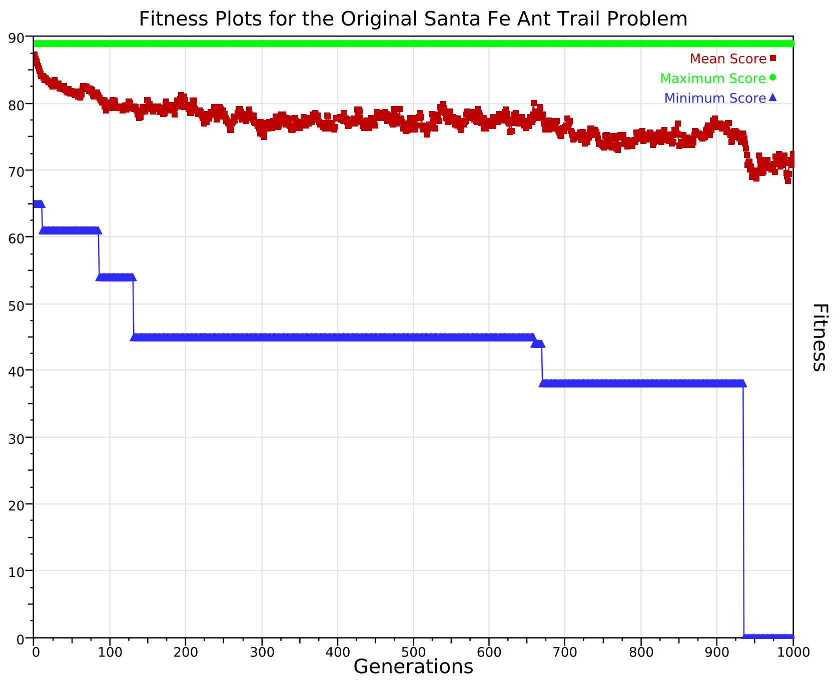


FIGURE 6.6: An optimisation run showing fitness by generation of the Santa Fe Ant Trail problem using the MOL code shown in Listing 6.7. As seen in this plot, a steady decrease in average fitness is followed by frequent drops in minimum fitness until the best possible score (0) is achieved.

For this single-threaded optimiser, it seems approximately 3.8 seconds is necessary to compile 200 random programs for the host processor. This does change however, depending on the structure of the candidates. For instance, if the tree of the program is sufficiently deep, many statements would need to be compiled, which will increase this time considerably.

The time taken to execute 200 random programs is approximately 1.7msec. The Predator-prey model discussed with regard to parallelism in the next section is far more computationally expensive, and requires careful consideration, as it is also stochastic to a certain degree. In the case of the Santa Fe Ant Trail problem, the result is deterministic, and only one agent is active, which makes it very suited to being computed on the host.

($\mathcal{I}$, p. 112) 6.5 Parallel MOL

It was observed that the performance of the optimising MOL language was quite prohibitively low, should the fitness evaluation phase be complex. Computing times vary, but are generally of the order of hours for a single experiment. Lessons learned from meta-optimisation (see Section 3.5) reaffirm that performance would be extremely problematic, mostly due to the averaging and re-averaging necessary to obtain a good fitness estimate (should the problem be nondeterministic), and that candidates are actually entire simulations. Van Berkel's effort was distributed across a set of processor nodes, but performance results given indicated that the execution of a single program took upwards from 350ms for a program of lowest complexity [283]; together with averaging, the author reported total runtime of around three hours for one experiment.

Stronger interest in large-scale agent-based models is also surfacing, where a large number of agents are simulated [220]. Being able to represent large populations is sometimes a necessity. For example, in ecology, a technique was even developed for approximating the influence of multiple agents in a "super-agent" [252]. When referring to a population of agents, of the order of 10^6+, it becomes impractical to use machine learning for developing models. At this point in time, even with aggressive code optimisations, evolving large systems is still out of reach, but efforts making use of GPU hardware bring this goal closer. Even if large systems are not used during optimisation, these can be scaled up afterwards; though, the dynamics of the system may change dramatically.

Fortunately, NVIDIA have released a backend for LLVM which generates Parallel Thread Execution (PTX) instructions for NVIDIA GPU hardware [165]. The implications of this is that any LLVM frontend can now (with appropriate modifications) generate the appropriate LLVM IR code suitable for compiling to PTX instructions. Terra [46], upon which MOL is built, is also capable of this. As explained earlier, agent-based model simulations implemented on GPU hardware have in the past involved custom code, or code transformations [244] from agent specifications such as the X-machine [37]. Also, previous implementations of Genetic Programming algorithms on GPU hardware were implemented in a way that candidates would be evaluated by using an interpreter [156]. Very sophisticated methods such as the evolution of CUDA PTX programs themselves in 2011 [41] using the CUDA driver API was perhaps a sign of what was to come with the NVIDIA LLVM backend.

In order to enable the lattice-based MOL language to be compiled for execution on GPU hardware, it is necessary to:

1. Adjust how the data is stored in the host C++ program
2. Support a population of models executing concurrently
3. Compile to a Terra function with a different signature

4. Generate CUDA code instead of host code
5. Reimplement a suitable source of random deviates
6. Consider different parallelisation strategies and how to implement these automatically
7. Consider the possibility of concurrency race conditions

Focus is given to the efficient computation of the objective function, in other words, the simulation of the candidate models so that fitness scores can be gathered quickly. In Chapter 4, genetic operators were themselves implemented in parallel to cope with large numbers of candidates. While this provided some additional good scaling characteristics, the computing of the objective function proved to be far more computationally expensive. The successful mitigation of which will surely dwarf the potential benefit that can be achieved by parallelising the genetic operators.

Previously, the lattice along with a temporary write-only lattice was allocated on the host. Should CUDA be enabled in a MOL model, the data is instead allocated on the GPU hardware by using the CUDA API [202]. These device pointers are provided to the Terra compiled function instead of pointers to host memory. The compiled MOL code is therefore able to operate on the lattice, as allocated by the host on the GPU hardware. The code parser and type checker are identical, but a separate CUDA code generator is used in order to accommodate the restrictions imposed by the CUDA GPU architecture. Compiled code is then mostly PTX instructions, wrapped with the necessary host code to launch CUDA kernels with the correct thread grid and block dimensions. Once a timestep is computed, the data is copied back from the GPU to the host and then passed to the visualiser.

To accommodate a population of different candidate models as opposed to simply simulating a single given MOL-implemented model, it is necessary to extend the visualisation module as well as allocating enough memory in the above-mentioned GPU memory for n separate candidate models. In essence, the separate portions of the allocated memory represent independent models, which are handed to their corresponding optimiser-modified MOL programs.

Compiling Terra code for CUDA is straightforward, provided that the boundaries of the device in terms of memory and thread resources are respected. The usual Terra code generated is essentially compiled into a single CUDA kernel, which is launched with a grid and block configuration, and its arguments, by a separate host Terra function. Given that an appropriate grid and block must be provided, this presents an opportunity to discuss different parallelisation techniques.

Three parallelisation strategies are implemented from which the user may freely choose. The first is a simple "one-thread, one-model" (1T1M) strategy, where a single CUDA thread is assigned a candidate model. This CUDA thread is then responsible for executing the entire model simulation once per time step. This is unsuitable most of the time, especially when one candidate model operates on a larger lattice, or the model is demanding of processing time required. The second strategy is named "one-block, one-model" (1B1M), in which an entire CUDA block is dedicated to computing a single model simulation once per timestep. While this may seem the obvious choice in nearly all circumstances, the limitations of block sizes (1024 threads maximum at the time of writing), mean that the lattice sizes have a limit. A great many candidate model simulations can be executed concurrently at reasonable speeds using this, but the limitation in lattice size is a considerable issue. The third strategy is termed "many-blocks, one-model" (*B1M), where multiple blocks are assigned to a single candidate model. This allows much larger model sizes, but race conditions become more difficult to eliminate, which requires further strategies.

The user must choose a strategy to overcome potential race conditions in a model simulation. Two strategies are provided from which the user may choose for the 1B1M and *B1M parallelisation strategies. The first is a "checkerboard" (or red/black) update scheme, which is common in lattice-spin models such as the Ising model [98]. Using this method, cells are computed in a way such that the nearest neighbours of any cell operated on are updated separately and independently. Using this "checkerboard" update, the Moore neighbourhood [94] can still cause some race conditions, and as a result, the user can also choose an additional "coarse" checkerboard update. Essentially the checkerboard is modified such that only $1/3$ is updated at one time. For one complete update, the grid of threads are offset so that each thread updates a 3x3 grid of cells. The Moore neighbourhood of these cells are therefore readable and writable without race conditions.[5]

Another considerable issue is the source of random deviates on the GPU hardware. There are various methods to accomplish this [161]. In this case, this was implemented by maintaining a separate GPU array with three unsigned long integers for every candidate model. These integers represent the u, v and w parameters of the Ran random number generator [229]. They are initialised by the host before being copied to the device. The host computes a new Ran state for each candidate model using a master Ran for providing a seed. This allows the MOL code to use as many random deviates as it needs to operate, since Ran also provides a colossal period (approximately $3.138(10^{57})$ [229]). To generate a random number, code is automatically generated from a macro function to update the Ran state and compute a random deviate of the specific thread.

6.6 Selected Results

Two experiments are conducted to evaluate the convergence and computing performance of the additional architecture introduced in this chapter. The first is The Santa Fe Ant Trail problem (see Section 4.1.2) which Koza solved using Genetic Programming [149]. This problem was approached using a select recombination structure (as in Section 6.4.3, in order to evolve the appropriate decision tree (expression tree) for an ant to collect all food placed on an irregular trail. The second experiment pertains to a more classical model: the Predator-prey model [104, 257] (see Section 2.1.3). The objective in the Predator-prey model is to evolve a list of ordered rules which are most suitable for the predators to catch the prey.

The original Santa Fe Ant Trail problem solved by Koza [149] used a ProgN3 nonterminal, which was more true to the original Santa Fe Ant Trail problem. In addition, multiple move commands can be issued in one timestep. To more closely resemble this, the MOL recombination code was altered and the ProgN3 nonterminal was introduced. The implementation of the Santa Fe Ant Trail was also modified to allow more than one move command per time step. In allowing the ant to make multiple movements in one timestep, the conceptual fitness landscape becomes less deceptive. A more gradual descent to an optimum was observed than that shown in Figure 6.6. It should be noted that the frequency of candidates which attempt to iterate over the entire lattice increased dramatically.

The parameters used for these experiments are shown in Table 6.3.

6.6.1 Santa Fe Ant Trail

($\mathcal{I}$, p. 71)

The fitness plot for this experiment is shown in Figure 6.7. There is a gradual (though small) decrease in mean fitness up to generation 120, and followed later by a slight drop around generation 280. The minimum fitness does decrease over time, indicating progress in the search. Though, during this run, the optimum fitness (zero) was not achieved, a number of semi-suitable programs were generated. It is worth noting that no generated program can achieve maximum fitness by brute force iteration of the entire lattice due to the 400 time step maximum [149].

[5]For more information on this coarse checkerboard, see Section 7.2.1.

Santa Fe Parameter	Value
Population Size	500
P(mutate)	0.2
P(crossover)	0.8
Program head length	12
Timesteps per Generation	400

Predator-prey Parameter	Value
Population Size	500
P(mutate)	0.2
P(crossover)	0.8
Timesteps per Generation	200
Generation Repeats	3

TABLE 6.3: Optimisation parameters used for the Santa Fe Ant Trail problem and the Predator-prey model.

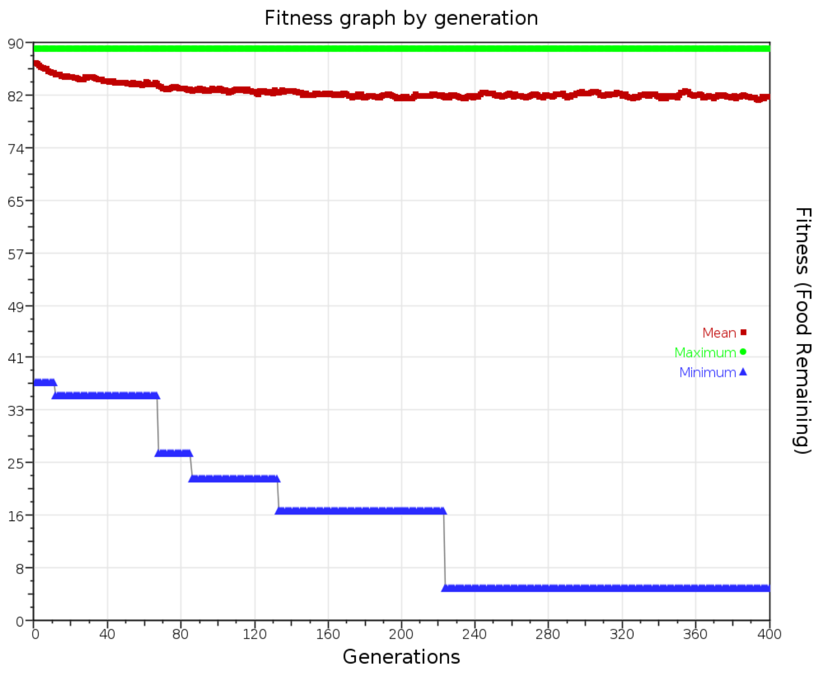


FIGURE 6.7: Fitness plot by generation for the Santa Fe Ant Trail problem computed by the host using the single-threaded version of MOL.

After the MOL program is first parsed by the Terra parser, and a typed syntax tree is constructed, it is duplicated 500 times, and the MOL optimiser modifies these trees. The time taken to compile all 500 typed Terra trees to *host* code (in single-threaded MOL) is approximately 14.8 seconds (averaged five times). Whereas, the time to compile 500 unique typed trees to PTX is approximately 473 seconds. This is clearly an undesirable amount of time, considering that this dwarfs the evaluation time.

The time taken to compute a single timestep of a system with 500 candidates is approximately 3.57msec (averaged five times) for the single-threaded host version, whereas the GPU-parallel version computes a frame in 2.5msec (averaged five times). These are both of the order of a second for evaluating the entire population in the current generation. Additional scaling data for increasing population sizes are shown in Figures 6.8(a) and 6.8(b). In these plots it is clear that unless there is excessive computation necessary in the evaluation of a population of candidates, then it is likely that the GPU version of MOL is not necessary. At this point it is not clear why there appears to be an exponential rise in compute time required for compiling larger simulations at runtime for GPU, though the use of CUDA RTCG in Terra is experimental at this point. Also interesting in the timestep plot is that the GPU code surpasses the CPU code at relatively small population sizes. Though the CPU code is faster for population sizes of 32 and less, it was expected that the CPU code will surpass the GPU until at least 128 candidate programs. Altogether, the total lattice size operated on for the largest population was sized $512(32) = 16384$ by 32 lattice sites. This is a colossal 524,288 lattice sites, which makes the computing time for one frame considerably more reassuring for both CPU and GPU code.

($\mathcal{I}$, p. 18)

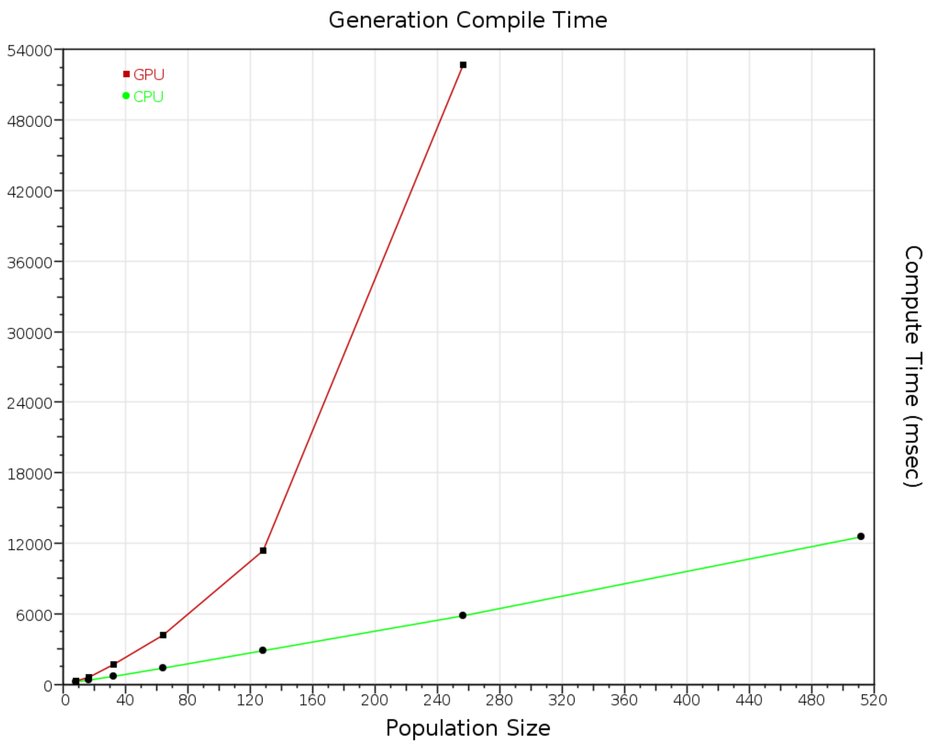
6.6.2 Predator-Prey Model

For the Predator-prey model, timestep computation times were measured for the single-threaded and GPU-parallel compilations for different population sizes and lattice sizes. This model is more computationally expensive to compute per timestep than the Santa Fe Ant Trail. The reason for this is that both predators and prey must execute a MOL program, meaning that at times, every lattice site will execute a program, whereas in the Santa Fe Ant Trail, there was only one program being executed, that of the ant.

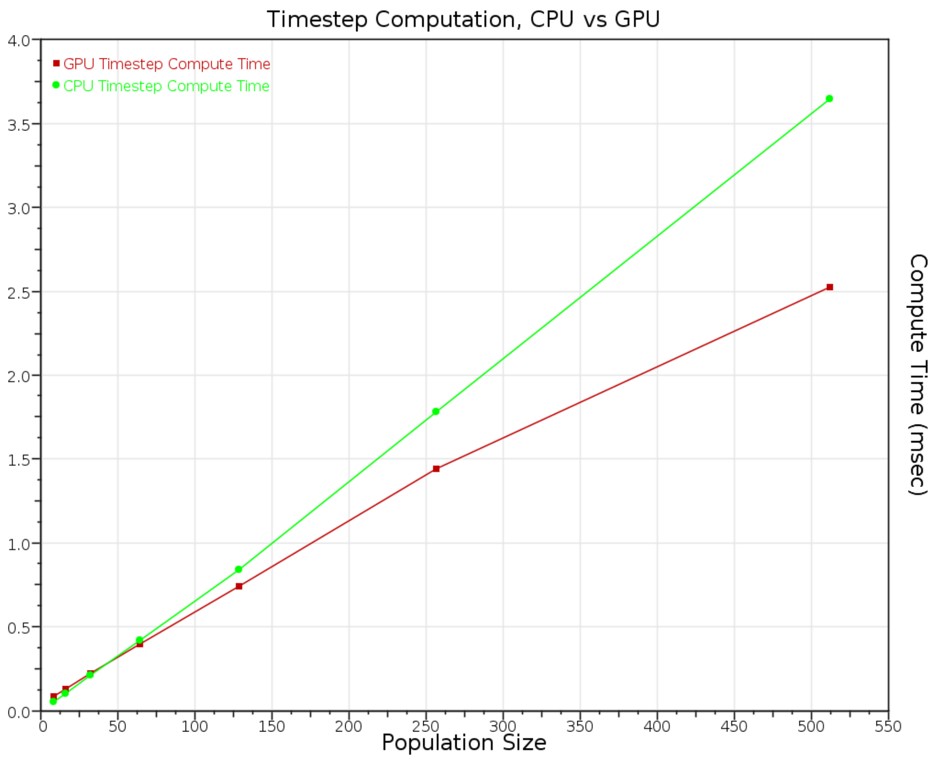
The optimisation objective is to select a permutation (with replacement) of rules for prey, to maximise the number of prey. Usually, the evasion strategy for prey is to find the closest predator, and move directly away from it. This is a simple but effective strategy. Emphasis here is given on performance results instead of convergence results.

The measured data in log-linear plots are shown in Figure 6.9. Different configurations of candidate model lattice sizes (8x8,16x16,32x32,64x64 and 128x128) were used, along with the “one-block, one-model” (1B1M) CUDA parallelisation strategy, the “many-blocks, one model” (*B1M) strategy, as well as single-threaded CPU configurations. The largest candidate model lattice (128x128) proved prohibitively expensive to compute by the single-threaded CPU configurations. There exist rapid increases in computing time between candidate lattice sizes, but increases in population size of these are relatively slower. The mere ability to simulate a lattice of size 128x128 with 64 heterogeneous candidates is enormously encouraging, considering that a population of this size is advantageous with regard to population-based optimisers. Unfortunately, while timestep computation scales well, compile time does not, however.

For the complete program shown in Listing 6.2, compile times from a MOL typed syntax tree to Terra code takes between 25msec and 30msec, however, some programs take up to 130msec to compile for the predator-prey model with heterogeneous programs (caused by the optimiser exploring the search space). The CUDA kernel compile time for 32 heterogeneous programs is approximately 40 seconds. For a trivially simple program (`mol move left end`), compile time from MOL typed tree to Terra is approximately 1msec, and kernel compilation is 530msec for 32



(a) Generation population compilation. Data point for GPU at 512 population size not shown: 539000msec. Parallelisation strategy used was "one-block, one-model."



(b) Timestep computation.

FIGURE 6.8: Performance plots of timestep computation and population compiling between CPU and GPU compiled MOL code. Data is averaged over the first 300 time steps of randomly initialised runs.

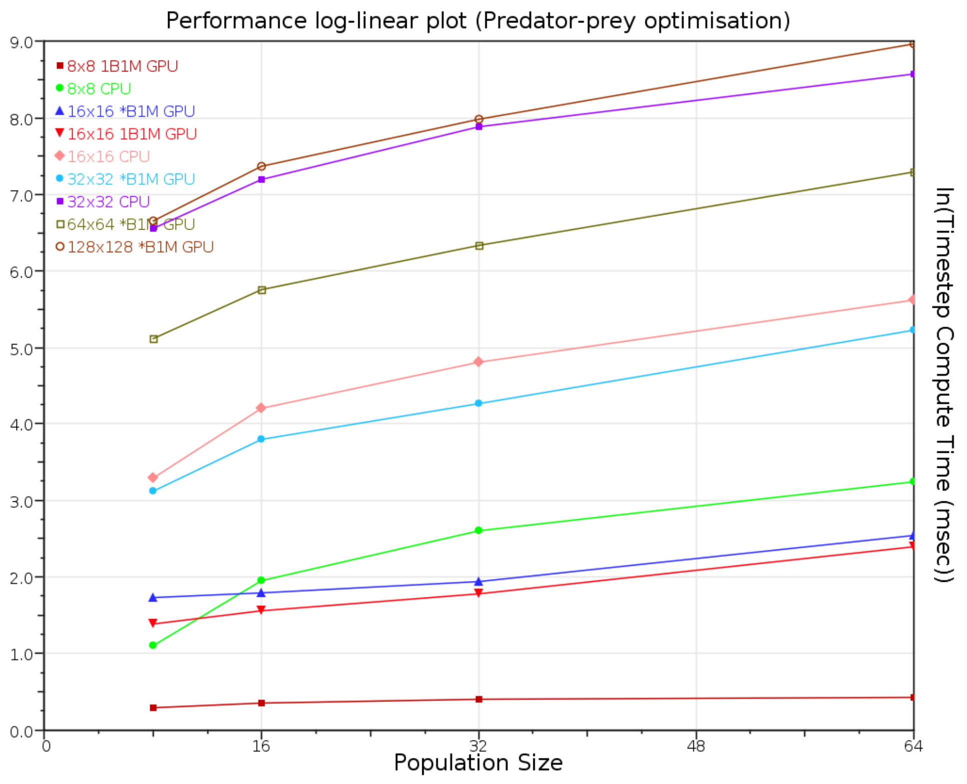


FIGURE 6.9: Performance plot for different system sizes both in candidate model lattice sizes and number of candidates in populations for both the parallelisation strategies “one-block, one-model” (1B1M) and “many-blocks, one model” (*B1M). CUDA block sizes in *B1M were restricted to 16x16.

identical programs. A slightly more complex trivial program, but one still containing an optimisation construct such as the following takes on average 5msec to compile from a MOL typed tree to Terra:

```
mol select recombination to minimise(1) move left end end
```

6.7 Discussion

The use of run-time code generation is a good method for improving performance, provided that the evaluation phase of a population of candidate models is sufficiently complex. For models in which evaluation is less expensive (such as the Santa Fe Ant Trail model), it is more appropriate to use the single-threaded version of MOL. This allows programs to be compiled faster, while suffering a very small drop in timestep computation performance.

Systems larger than 512 candidate models appear to be out of reach of GPU run-time code generation. Such a result was expected, given that Cupertino et al. chose to evolve PTX code itself rather than use the CUDA run-time library to compile C code [41] noting that the latter would be too computationally expensive.

These limitations reaffirm that it is unwise to ignore the power of the newer multi-core processors available [36]. At the same time, the GPU should be applied when most or all of its theoretical computing power can be achieved. This demands proper choice in parallelisation strategy, which is anticipated to be automatically selected using relevant model information in the future. It is certainly possible to generate multi-threaded code on the host processor instead of CUDA code. This is a promising area for future work.

The next chapter makes a more thorough study using a lattice-based photobioreactor model. A thorough discussion on the implementation of such a bioreactor using ABM is given, and comparisons with C++, CUDA, and MOL are made and discussed. Some experiments in selective model induction are also conducted.

”...essentially, all models are wrong, but some are useful.”

George E. P. Box

CHAPTER 7 PHOTOBIOREACTOR MODELLING

Biological processes such as growth kinetics [19] are difficult to model with precision. The numbers of cells under consideration are often excessive, sometimes ranging between 5 and 6 million cells per mL [235]. This has led to an increased emphasis on differential equations for understanding the process [181], as opposed to stochastic individual-based simulations on cellular level. The advent of affordable and high performance graphics hardware has allowed larger system sizes to be scrutinised, but there is still room for improvement. Agent-based Modelling (ABM) is used in this chapter, due to its natural application to this field. ABM is best used in situations where a system can be naturally divided into small discrete decision-making entities, which is entirely applicable in this instance [215].

This chapter carefully considers the problem of stochastically modelling algaculture with some aimed accuracy, particularly the growth kinetics associated with such systems. Some models in the literature are briefly considered, and a new model is proposed. The purpose of synthesising this model is to provide an additional test bed for complex individual-based models, where parameter optimisation is often not enough for computationally inducing an appropriate model. The MOL language presented in the previous two chapters is used to implement the model and compare against CUDA and C++ implementations. Examples of structural optimisation are also given and discussed.

This chapter extends upon work previously published by the author in *Modelling and Simulation in Engineering*[1], the *Proceedings of the 14th IASTED International Symposium on Intelligent Systems and Control (ISC 2013)*[2], and also the *Proceedings of the 14th International Conference on Bioinformatics and Computational Biology (BIOCOMP13)*[3].

7.1 Introduction

ALGAE IS DELIBERATELY CULTURED FOR SEVERAL REASONS, such as carotenoid production [74], biodiesel [158] (though in infancy), Spirulina nutritional supplements [243], and effluent treatment [194]. Each of these applications involve a different species of algae. Cultivating these strains is not always trivial,

[1]A. V. Husselmann and K. A. Hawick. Simulating growth kinetics in a data-parallel 3D lattice photobioreactor. *Modelling and Simulation in Engineering*, 2013

[2]A. V. Husselmann and K. Hawick. Intelligent individual agent-based simulation of photobioreactors and growth control. In *Proceedings of the 14th IASTED International Symposium on Intelligent Systems and Control (ISC 2013)*, 2013

[3]K. A. Hawick and A. V. Husselmann. Photo-penetration depth growth dependence in an agent-based photobioreactor model. Technical Report CSTN-204, Computer Science, Massey University, Auckland, New Zealand, Las Vegas, USA, July 2013

especially in the case of *Haematococcus pluvialis*. This algae is used to naturally produce a valuable carotenoid known as Astaxanthin: a pigment found in certain aquatic animals [90] and often used for pigmenting salmon and trout [74]. It is also a powerful antioxidant [235]. To induce carotenogenesis (the production of carotenoids), it is necessary to incite growth to reach a suitable culture density, and then stress the culture by way of salinity or excessive illumination. This causes the culture to form Haematocysts which accumulate Astaxanthin, which can then be extracted by centrifuging and drying the biomass [74].

There are also different methods of cultivating algae, some of which include: outdoor ponds, column photobioreactors, flat-plate photobioreactors and tubular photobioreactors [281]. In addition to these, there are also airlift bioreactors [35].[4]

Each of these methods have disadvantages and advantages. In this chapter, a focus is given to bubble-column photobioreactors. These are common for temperature and light controlled indoor production facilities. Nutrients are fed to the reactor by bubbling gas such as nitrate or phosphate through a manifold in the bottom of the column named the sparger.

Regardless of reactor geometry and location, illumination is extremely important. Among the factors which influence the growth kinetics of algae, light is the most important [211, 181, 74]. For this reason, much focus is given to modelling illumination in the reactor. The major factors related to illumination which affect photosynthesis in aquatic ecosystems include light scattering and absorption [144] as well as mutual cell shading [181], which collectively cause photo-limitation or photo-inhibition in the absence of light or low photon flux density [57].

Microbiological simulations are by no means new. There have been several simulations in the past of microbiological phenomena, which have (for the most part) been collectively known as individual-based models [65, 87, 233]. Particularly prominent among these are bacterial growth simulators, such as BacSim [153] and BSim [83]. These simulators are suitable for a wider audience and hence stimulate cross-discipline inquiry in this area. Individual-based Modelling (IBM) is closely related to ABM, and hence, they have much in common. One example of a simulator for algal reactors is the work of Greenwald, who proposed a stochastic simulation of a rectilinear photobioreactor (PBR) [85] in 1 dimension with Brownian motion [73].

In general, previous agent-based algal growth models typically assume a 1-dimensional lattice [85, 211]. This work focusses on a 2-dimensional lattice in an attempt to better model the local interactions caused by hydrodynamics in order to more accurately determine illumination history. Illumination history is important to the cell division rates in a culture [74, 181, 296], and factors such as fluid dynamics determine the effects of mutual shading between cells, as well as their exposure (or over-exposure) to the illumination source. In the simulation discussed here, this combination of photo-limitation, photo-inhibition and mutual shading determine the illumination history of a cell.

IBM has recently become a concept which has gained much interest in the field of microbiology in general [65]. In the past, IBM was only really applied to microbiology, and has very recently become a synonym for Agent-based modelling. Railsback and Grimm in their work of 2012 use the terms ABM and IBM interchangeably [233]. In that sense, models with autonomous entities have been developing concurrently in more than one field over the past few decades.

The objective of the modelling process in this case is to closely model the growth kinetics of *H. pluvialis*. A model of such a process would be immensely helpful to gain insight in to how to improve a production facility to increase the yield of the highly valuable carotenoid Astaxanthin, without brute force iteration. In order to produce Astaxanthin from *H. pluvialis*, it is necessary to first cultivate vegetative cells where no carotenogenesis takes place.

[4]This list is by no means exhaustive, however.

Once a suitable culture is achieved (high cell density), then the culture is stressed in some fashion. This is done by causing nutrient deficiency, or increasing temperature, or introducing NaCL into the medium. Haematocysts for example, develop 2-3 days after the culture has begun to be stressed. Another 3-5 days after this and these cysts would have accumulated between 1 and 3% Astaxanthin, and would be ready for harvest. Harvesting is a delicate process, where drying of biomass is done and centrifuging is used to separate out the desired products. In terms of productivity it is optimal if the bioreactor vessel is harvested as soon as possible.

Maximising the output of Astaxanthin can be done by inciting rapid cell growth to obtain a high cell density; from when the culture can be stressed to increase yield. In the literature, it is reported that the only aspect that changes growth rate considerably when the cells are not under significant stress from photoinhibition, photolimitation, nutrient deficiency, temperature stress, or shear-stress, is light availability [74, 235]. In a production facility, all of these factors would be carefully regulated, however, cells closer to the surface of the vessel (hence illumination source) tend to absorb more light and shade other cells. This is further complicated by bubbles from the feeding lines, which tend to move cells through the medium. The literature also reports that the ability of cells to grow and divide also further depends on the history of illumination on those cells [160]. This means that an approximation to fluid dynamics in such a reactor simulation could improve accuracy to some degree.

The overarching goal of this model is ultimately to represent the system as a set of individual, autonomous, interacting agents [65]. The process of doing so requires at least a set of behaviours for each agent and interaction specifications. Three important factors are taken into account. The first concerns some method of approximating hydrodynamic flow, and hence illumination history. The second concerns some method to determine when and how cell division occurs, and finally, some method to increase performance for larger system sizes.

The rest of this chapter is organised into a section first detailing the conventional approach to modelling a photobioreactor with the above objectives (Section 7.2), followed by Section 7.3 which describes the same modelling process using MOL, a language developed in the previous two chapters. MOL allows virtually automatic parallelism along with a structural optimiser which is experimented with in the context of agent-based modelling of algae in a photobioreactor.

7.2 Conventional ABM

($\mathcal{I}$, p. 11)

As mentioned, this section considers what is necessary to obtain an agent-based photobioreactor model such as that shown in Figure 7.1, which is capable of demonstrating the dynamics of algal growth, and more importantly, its associated growth kinetics.

Modelling in this section is accomplished using custom-written C++ and CUDA code. This serves as a base-level comparison against the technique discussed in Section 7.3 in terms of both utility and performance. Three aspects of the model requirements organised into three phases include the approximation of hydrodynamics and illumination, and secondly, cell division, and finally mutual cell shading. All three of these phases were implemented using C++ and Compute Unified Device Architecture (CUDA).

7.2.1 Hydrodynamic Flow and Illumination Approximation

The first phase of the modelling effort concerned the use of the Kawasaki site exchange model [141] for a baseline mass transfer approximation. The Kawasaki exchange model is reminiscent of the Ising model for ferromagnetic lattice simulations. Using a ferromagnetism model such as this is somewhat unorthodox; however, the lattice-gas characteristics of the Kawasaki exchange model [92] certainly indicates otherwise. Further, a probability of cell

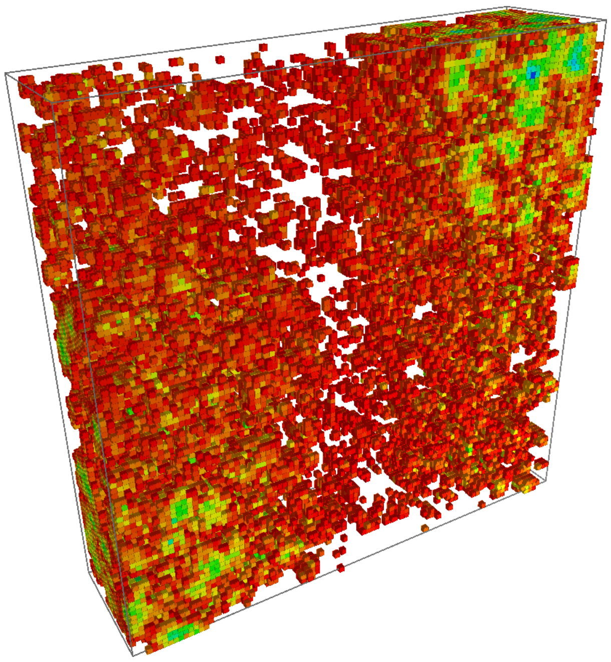


FIGURE 7.1: A visualisation of a simulated photobioreactor at a very late time step. Cubes represent algal cells.

division dependent on exponential decay of illumination intensity was used. This used the lateral distance from the closest of the left and right vessel walls. Additionally, a weak gravitational force was also used, which has been previously researched in the context of sedimentation simulation [92].

The Kawasaki model itself depends on energy minimisation and stochasticity. Essentially, the species in a lattice would diffuse until they make contact with other identical species. Contact between identical species is considered a bond, which requires energy to break. Exchanges occur between two neighbour species should a decrease in local energy result. Otherwise, a temperature parameter along with a random deviate and Metropolis probability is used to decide whether to accept an exchange, regardless of the bonds that may be broken as a result. A visualisation of this with a relatively low temperature is shown in Figure 7.2.

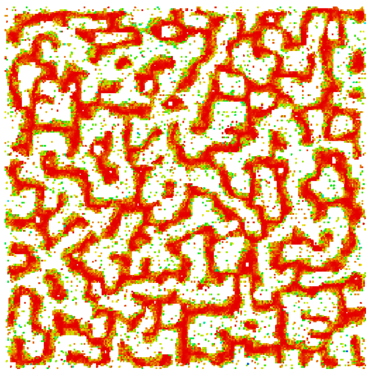


FIGURE 7.2: The canonical Kawasaki exchange model.

To facilitate some additional inquiry into the movement of cells, a site has an age counter, which is incremented should it be exchanged with a neighbouring cell. This results in the coloured lattice sites shown in Figure 7.2, which are normalised and spread across the hue parameter of the HSV colour space where red is close to zero. To conserve computing effort required (and allow compatibility with parallel MOL), both the species state variable (empty or occupied) and age counter are stored within the same 32-bit integer. The most significant byte is used for the state variable, and the first three bytes as the age counter. Empty lattice sites are denoted by 32 zero-bits.

Computing cell division is based loosely on the illumination upon the lattice sites, which shall be known as the *cells* henceforth. For modelling light penetration into the vessel, a standard exponential absorption function was used in the spirit of the Lambert-Beer law from the study of optics [155, 20]. Scatterance is another aspect to hydrological optics, in tandem with photo absorption [144], however, in this work, focus is given to absorption in the medium by a constant attenuance exponent (β). The equation describing the cell division probability $P(\text{split})$ is given in Equation 7.1.

$$P(\text{split}) = \gamma e^{-\beta f^2} \tag{7.1}$$

Here, f is the lattice width fraction of the site to the closest wall (shown in Equation 7.2, w is the width of the lattice). The γ variable is a simple amplitude variable, and finally, β controls the slope of the decay.

$$f = 1 - \left|1 - \frac{2x}{w}\right| \tag{7.2}$$

Gravitational bias is introduced by widening the probability that a cell will exchange into the site below it, should that site be chosen for a potential exchange; as well as narrowing the probability that the cell will exchange into a site above it. The motivation behind this is that in the interest of energy minimisation, a site below a cell is regarded as a drop in energy, and *vice-versa* with the upper site. At this point, 2-dimensional lattices are used for explanation purposes.

To simulate heavier cells due to nutrient absorption, the gravitational bias was modified to accept influence from the cell age counter, which was assumed to be proportional to nutrient absorption. This assumption, while somewhat arbitrary, gives the effect of older cells sinking to the bottom of the vessel. The immediate effects of this on the original Kawasaki site-exchange model is sedimentation, which is clearly seen when multiple species are present in the lattice [92]. Figure 7.3 shows the effects of a weak gravity force in the Kawasaki model [92] with multiple species in the lattice.

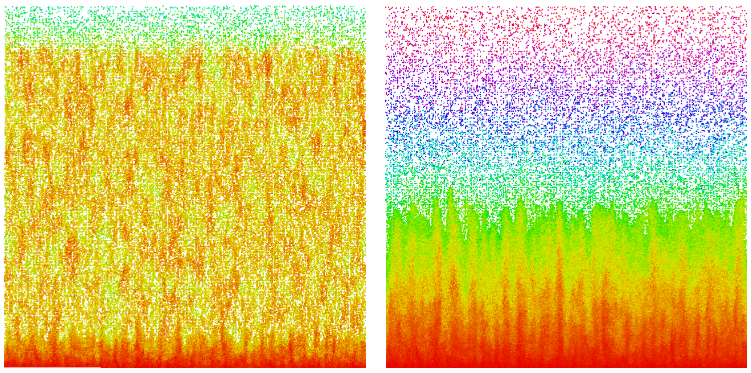

(a) An early timestep screenshot. (b) A later timestep screenshot.

FIGURE 7.3: Kawasaki model with weak gravitational forces at different timesteps during simulation.

Larger sytems and more temporally distant system states can be attained by using Graphical Processing Units (GPUs). The use of NVIDIA's CUDA is discussed in Chapters 2, 3 and 4. Due to the larger neighbourhood of cells that must stay in synchronisation, a "checkerboard update" is used to ensure that threads do not cause race conditions. In the Kawasaki exchange model, cells in the lattice must be able to read/write in the Von Neumann neighbourhood of Manhattan distance $r = 1$ from the cell index, and at least read from the extended Von Neumann neighbourhood of Manhattan distance $r = 2$ [92]. This imposes some restrictions on the order the lattice cells are processed. Sequential simulations typically overcome this problem by performing Monte Carlo updates of cell sites. This work follows a similar path, except random order updates are performed in a 3x3 grid around each one lattice site, and the CUDA thread grid is divided into a smaller lattice able to process $1/3$ of the lattice at a time. This ensures that there is a gap of two cells between each concurrently running thread. The process for this simulation is outlined in Algorithm 15.

```
initialise sites empty with $P(\text{empty}) = 0.5$
**for all** time-steps **do**
  **for all** 3x3 blocks in lattice **do**
    **for all** 9 sites $i$ in each block, random order **do**
      choose a random neighbour site $j$
      compute energy change if $i, j$ exchanged
      **if** energy falls **then**
        accept change and do exchange
      **else**
        compute Metropolis probability $p$
        add gravitational bias
        obtain random probability $r_1$
        accept change conditionally on $r_1 < p$
        compute cell division probability $P(\text{split})$
        obtain random probability $r_2$
        divide on $r_2 < q$
      **end if**
    **end for**
  **end for**
**end for**
```

ALGORITHM 15: A GPU-parallel Kawasaki model algorithm with simple cell division.

An interesting effect of the Kawasaki model is that of multiple species. By increasing the number of species, it is possible to see the effects of competing growth. This was accomplished by "inoculating" the bioreactor vessel at the edges with different species. This data was incorporated into the 32-bit integer by reserving a few of the most significant bits. The ability to do this may be of interest, considering the very important task of ensuring that a bioreactor is not contaminated with foreign algal or bacterial strains. Being able to observe the effects of a foreign strain could be useful for detection purposes. Results of this are shown in Figure 7.4.

Some qualitative and quantitative results gathered from the simulation described by Algorithm 15 are shown in Figures 7.6 and 7.5. By gathering such data, it is possible to gain a sense of the simulation dynamics and more clearly notice limitations and shortcomings when compared to expected growth rates given certain lighting conditions.

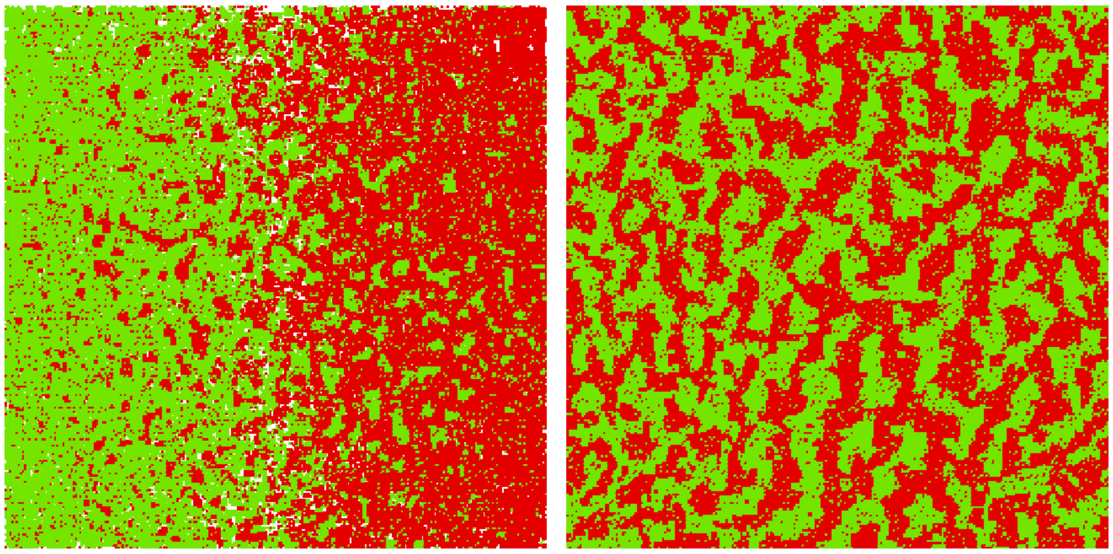

(a) Two competing species at an early timestep. (b) Two competing species at at apparent equlibrium.

FIGURE 7.4: Competing species in a Kawasaki site-exchange model.

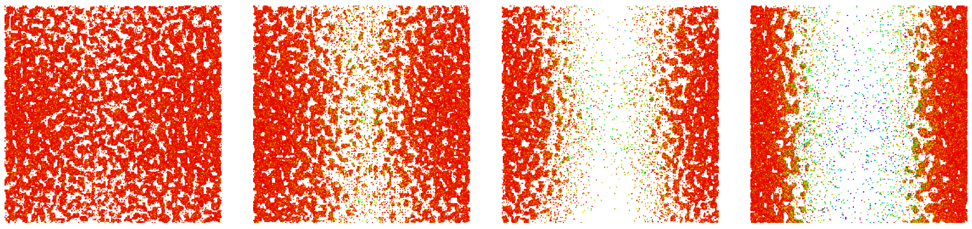
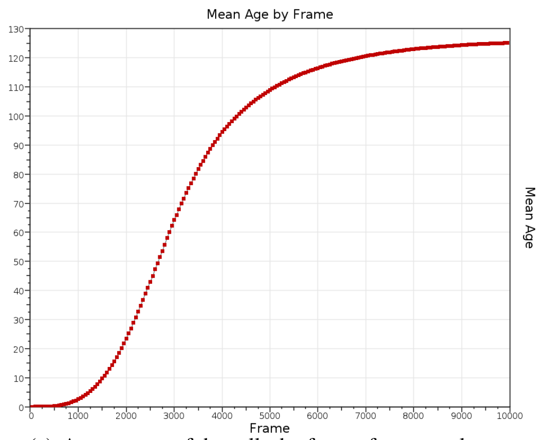
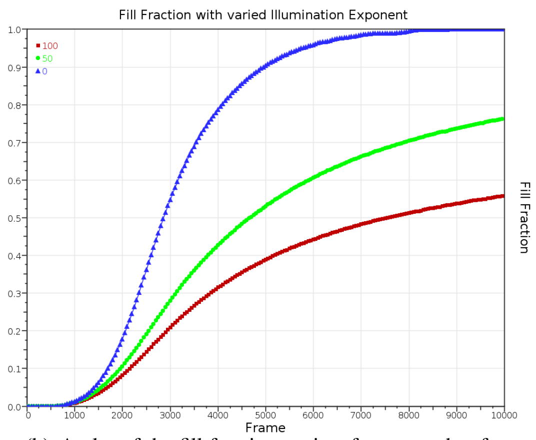


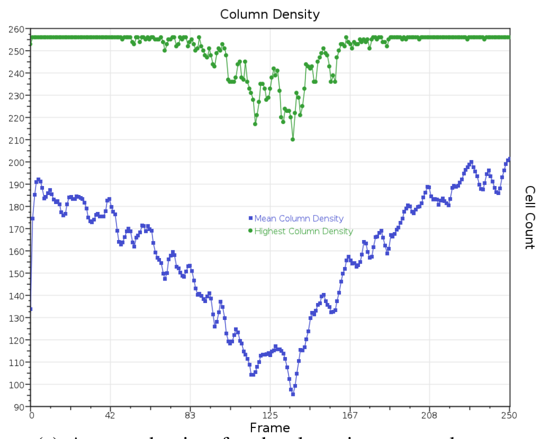
FIGURE 7.5: Visualisation of culture growth stagnation at illumination exponent values of 0, 20, 50, and 100 in order from left to right for sample runs. It is interesting to note that although 50 and 100 appear similar, they have different spatial densities.



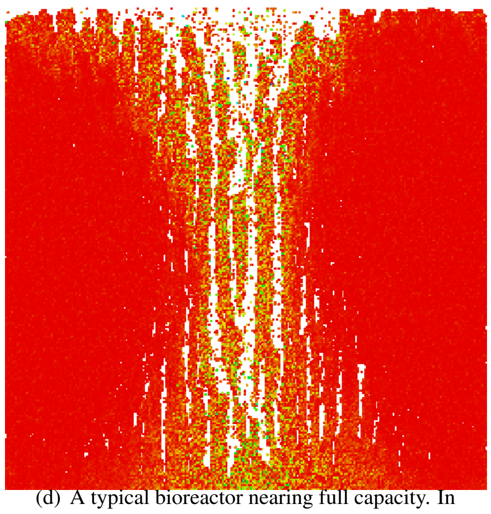
(a) Average age of the cells, by frame, for a sample run with $T = 0.4$ and $\beta = 15$.



(b) A plot of the fill fraction against frame number for differing values of the decay exponent parameter β. The data in this plot have been average over 100 independent runs for each frame.



(c) Average density of each column in one sample run.



(d) A typical bioreactor nearing full capacity. In this sample run, the influence of gravity was set 10 times higher to make the effects more clear.

FIGURE 7.6: Qualitative and quantitative results gathered from the first phase in agent-based modelling of the algal photobioreactor.

The average ages of cells during a typical simulation run is shown in Figure 7.6(a). Due to the small inoculation at either side of the vessel, it is expected that the first thousand frames have low ages. This is followed by a period of rapid growth, where cells are constantly moving, and ages are increasing rapidly. Finally, when the vessel is near capacity, ages reach approximately 125 and stagnate as less movement is possible. The temperature parameter was at a constant 0.4 during this experiment.

A fill fraction plot by timestep for the vessel is shown in Figure 7.6(b). The data was averaged over 100 independent runs, and for three different values of the β illumination exponent $(0{,}50$ and $100)$. The exponential drastically reduces the probability of a cell division towards the centre of the tank for values less than 50.

In Figure 7.6(c), the column density is shown on the lattice as an average, and maximum. These were averaged over a single sample run. As can be seen clearly, density is much less in the centre, even at the maximum measured. The variability in the data is due to the stochasticity of the algorithm. Averaging the results over separate runs as well will yield a smoother curve, however this was simply intended to cast light on the spatial configuration of cells.

Figure 7.5 shows the states in which the photobioreactor's growth stagnates due to insufficient illumination. The different values of β used to generate these images were $\beta = 0$, $\beta = 20$, $\beta = 50$ and $\beta = 100$. Smaller values of β (left two images) cause more cell division in the centre of the tank, whereas larger values cast darkness over the centre of the tank. While cells are able to survive in the centre of the tank, the probability that they divide is much lower.

Though there seems to be a growth rate attained which is loosely reminiscent of actual growth rates, important factors such as photoinhibition and mutual cell shading are ignored in this prototype. Cells do shade one another from the source of illumination to some extent, which is difficult to model in a discrete lattice simulation. In this case, an exponential light decay was used through the medium, assuming constant density. This may not always be the case. Instantaneous differences, particularly when the culture is relatively young, may have a significant impact on culture growth rates.

7.2.2 Cell Division

Having established a very simple 2D bioreactor model in the previous section with a rough approximation to hydrodynamics and cell illumination history, this section focusses on providing a more accurate cell division system. The combination of the Photosynthetic Factory Model (PSF) [57, 58] with the Kawasaki site-exchange model is presented and discussed in this section.

A thorough discussion of the PSF model is out scope. However, a brief summary is provided for self-containment. The PSF model facilitates a simplification of the photosynthetic process by considering it as a Markov process [57]. Three discrete states are allowed: activated, resting and inhibited. Cells may transition between these states with certain probabilities. The state diagram is shown in Figure 7.7. Modifications added are simply the addition of cell division and increment states. The former allows the cell to split depending on $P(\text{split})$, and the latter increments a counter. The counter is used to influence the growth rate, and is described later.

State transitions depend on the probabilities specified in the diagram shown. Parameters were hand-selected for P_α, P_β, P_γ and P_δ (the symbols in the state diagram are subscripted to avoid conflicting use) to 0.7, 0.5, 0.3 and 0.4 respectively. A cell may divide if it is in the resting state, and depending on $P(\text{split})$, which is discussed below. In order to observe the effects of photo-inhibition and limitation, the counter (now the activation state counter) is modified to measure how long a cell spends in the activated state. This is then used to add a small contribution on to the cell division probability, $P(\text{split})$, which is also discussed below.

The changes to the process shown in the previous simulation (Algorithm 15) are shown in Algorithm 16.

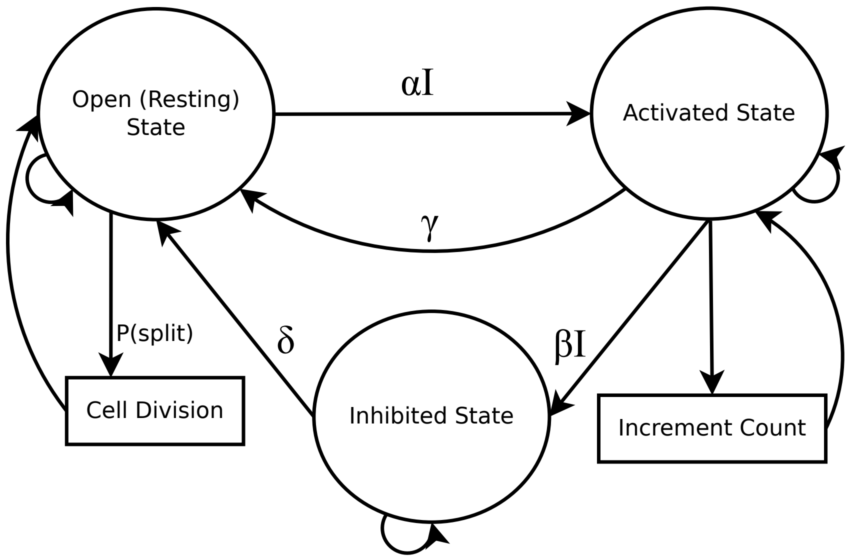


FIGURE 7.7: State diagram of the Photosynthetic Factory model and additional flow.

```
$N = L^2$ for square lattice
blank lattice, initialise 10 random cells
for all time-steps do
  for all 3x3 blocks in lattice do
    for all 9 sites $i$ in each block, random order do
      choose a random neighbour site $j$
      compute energy change if $i, j$ exchanged
      if energy falls then
        accept change and do exchange
      else
        compute Metropolis probability $p$
        obtain random probability $r_1$
        accept change conditionally on $r_1 < p$
      end if
      if in activated state then
        increment activated state counter
      end if
      if in resting state then
        compute cell division probability $P(\text{split})$
        obtain random probability $r_2$
        divide on $r_2 < P(\text{split})$
      end if
      state transition
    end for
  end for
end for
```

ALGORITHM 16: Kawasaki Exchange Monte-Carlo Algorithm including the PSF model.

Computing $P(\text{split})$ is modified to include a contribution from the activated state counter. The effect of light intensity decay is still included. The new formula for computing this quantity is shown in Equation 7.3. This equation shows the probability of a cell dividing where f is the distance to the nearest wall of the reactor vessel. In this equation, β is set to a constant 15, unless noted otherwise. The second and third terms in the equation are the effects of the activation state counters. This is added to ensure that the time a cell spends in the activated state corresponds to a slightly higher cell division rate [211]. The curve this produces against a is logarithmic with a y-intercept of zero.

$$P(\text{split}) = \gamma e^{-\beta f^2} - \ln(\frac{a}{40} + 2)^{-1} + (\ln 2)^{-1} \tag{7.3}$$

An additional modification from the initial design presented in the previous section is that the medium is now inoculated by choosing a set of random sites. This method attempts to mimic the effect of injecting a sample of a strain of algae into a well-mixed medium. The system state after such a sample inoculation some 100 timesteps from $t = 0$ is shown in Figure 7.8.

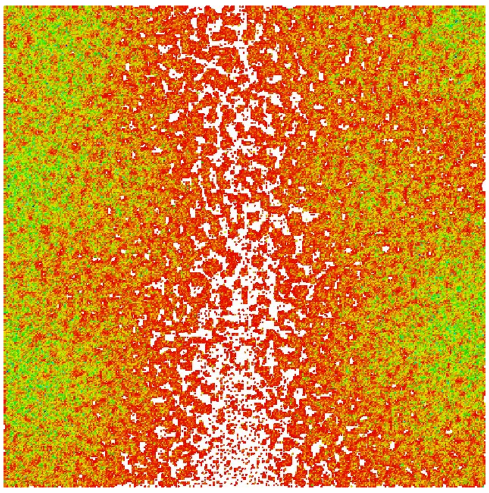


FIGURE 7.8: Model configuration showing photo stimulated agent preferential growth at right and left of simulated bioreactor.

A plot of the activation state counters are shown in Figure 7.9. The plot shows the highest and lowest counter, as well as a standard deviation and mean. The data shown has been averaged across 100 independent runs.

Modifying the light decay parameter γ has a dramatic effect on the fill rate of the bioreactor. Several curves for different values of γ are shown in Fig. 7.10. Each curve was averaged over 100 independent runs. The horizontal line through the curves at the centre indicate the point ($t_{1/2}$) the half-life of the system is considered to have been reached. The quantity $t_{1/2}$ is defined here as the number of time steps taken to reach a fill fraction of 50% in the bioreactor.

As can be seen from the fill fractions (growth curves), the maximum of 100% fill fraction is reached logarithmically. In practice, bioreactors are harvested after having reached a suitable culture density, which could perhaps be described as a fill fraction from 40% to 60%, considering that the lattice the simulation is built on is discrete. It may therefore be useful to compute the time that is likely needed for the reactor to fill to a certain level. This "half-life" of the system $t_{1/2}$ can be determined from the data collected.

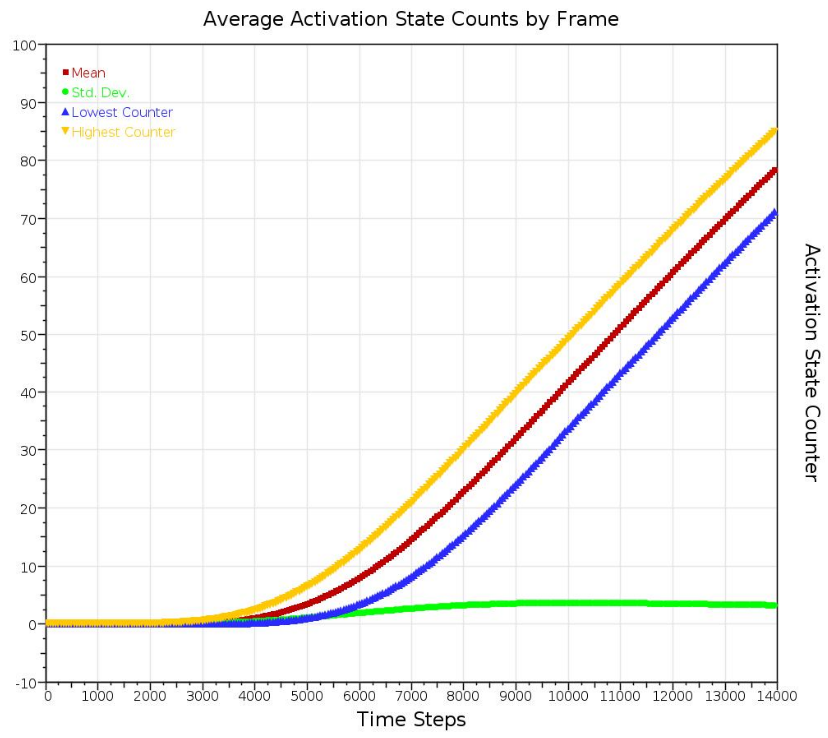


FIGURE 7.9: Plot of the activation state counters by time step. The data shown is averaged over 100 independent runs.

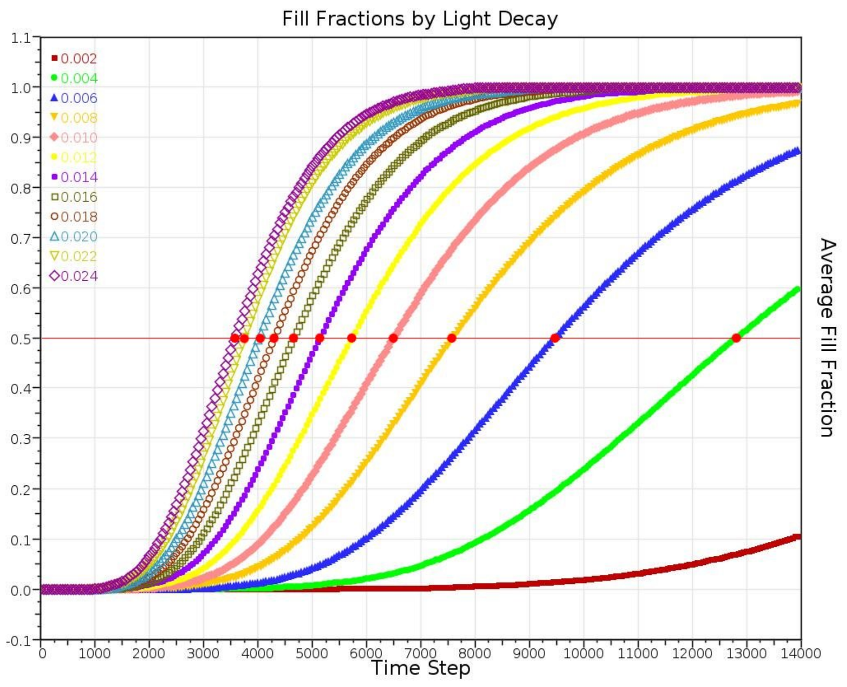


FIGURE 7.10: Plot of fill fractions by frame and decay parameter, averaged over 100 independent runs.

Plotting the light intensity parameter values γ against $t_{1/2}$ obtained in the system yields the plot shown in Figure 7.11. This plot suggests a relationship between $t_{1/2}$ and γ shown in Equation 7.4. Each point in this plot is averaged over 100 separate runs, and the error bars represent the standard deviations of the points.

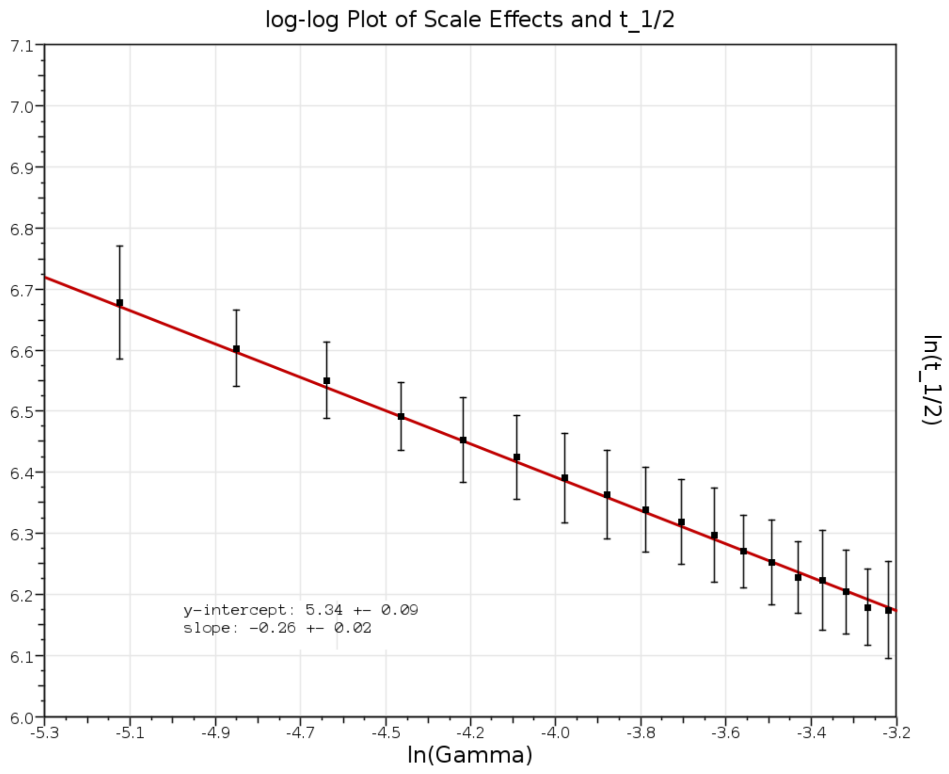


FIGURE 7.11: Log-log Plot of the brightness scale parameter (γ) against the time steps taken to reach a fill fraction of 0.5 ($t_{1/2}$).

$$t_{1/2} \approx e^c \cdot \gamma^m \tag{7.4}$$

The negative slope in the plot shown in Figure 7.11 suggests $m = -0.26 \pm 0.02$ and $c = 5.34 \pm 0.09$. As expected, increasing illumination intensity while keeping the exponential light decay in the medium can only reduce $t_{1/2}$ to a certain degree. The effects of mutual cell shading is still not considered, but will be discussed in the next section.

The light decay parameter β variation in Figure 7.12 appears to show a very weak linear relationship with $t_{1/2}$. Such a relationship reaffirms that lower values of β (corresponding to a more clear liquid with less light decay) improves growth rates. In practice, mutual cell shading is the major contributor in controlled environments to light decay in the medium. The use of the Lambert-Beer law for light decay in the medium in this sense, was simply approximating this mutual cell shading while crudely assuming that all cells are distributed equally throughout the medium.

It would appear that the amalgamation of the PSF model with the Kawasaki site spin exchange model provides loosely realistic growth kinetics in the simulation of a photobioreactor. However, negative cell growth is not accounted for (cell deaths), and also mutual cell shading is disregarded but approximated assuming that culture

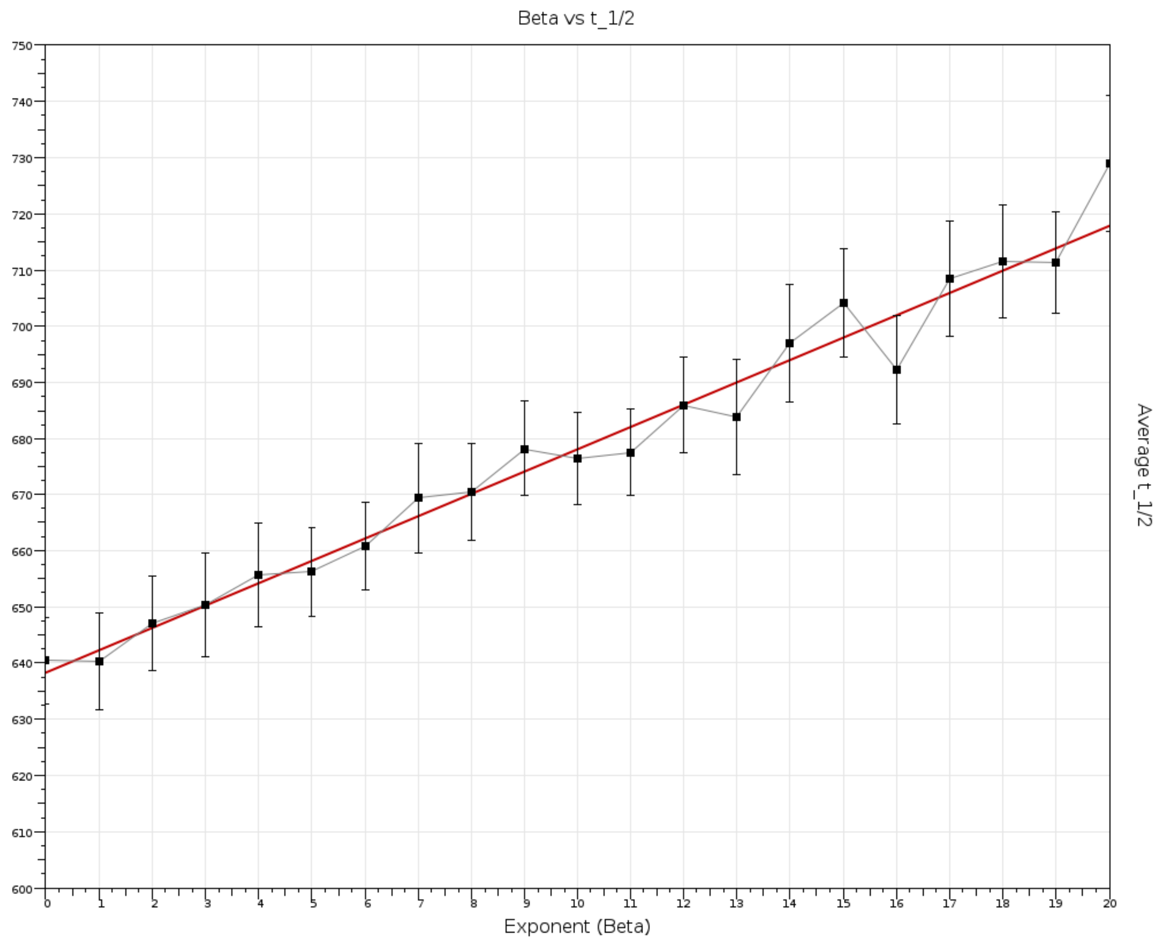


FIGURE 7.12: Plot of the (β) exponent against the time taken to reach $t = t_{1/2} = 0.5$. Here, each point is averaged over 100 independent runs, and error bars represent standard deviations. The γ value for these runs was 0.032.

density is uniform across the entire reactor. In the next section, these two assumptions are eliminated, and the simulation is expanded to 3 dimensions, so as to mimic the geometry of a flat panel photobioreactor.

7.2.3 Mutual Cell Shading and Negative Biomass Growth

As described earlier, the probability of cell division is key to the simulation, and depends directly on the number of state transitions a cell has had between activated and resting states. Illumination is indirectly important also, since the number of state transitions between activated and resting depend on how much illumination the cell receives, augmented by cell shading and decay. The new full equation describing illumination across the bioreactor vessel is shown in Eq. 7.5.

$$I = \gamma((1 - \frac{S_L}{L})e^{-\beta(x/L)} + (1 - \frac{S_R}{L})e^{-\beta(1-x/L)}) \tag{7.5}$$

Here, cell shading is represented by S_L and S_R. These are simply the number of cells between the current cell, and the left and right vessel walls. x is the position of the cell and L is the width of the reactor. β and γ provide attenuation and scale to the degradation of the light penetration.

Finally, the actual formula for $P(\text{split})$ is simply a scaled $d(a)$ from Eq. 7.6 using Equation 7.5.

$$d(a) = -\ln(\frac{a}{40} + 2)^{-1} + (\ln 2)^{-1} \tag{7.6}$$

This function simply tails off probabilities for larger values of a (state transitions from activated to resting) which can be between 0 to 200 in the experiments conducted here.

Figure 7.13 gives a visual representation of illumination from the left to the right sides of the vessel. $x/L = 0$ represents the left edge, and $x/L = 1$ the right edge. While this surface describes the general availability of irradiance on cells, it does not take into account the effects of mutual shading. Mutual shading applies a further fractional coefficient to the illumination, and its landscape is shown in Figure 7.14. The "Shading" axis represents how much of the vessel is obstructed for the cell under consideration. The inflection at $x/L = 0.5$ divides the left edge from the right edge.

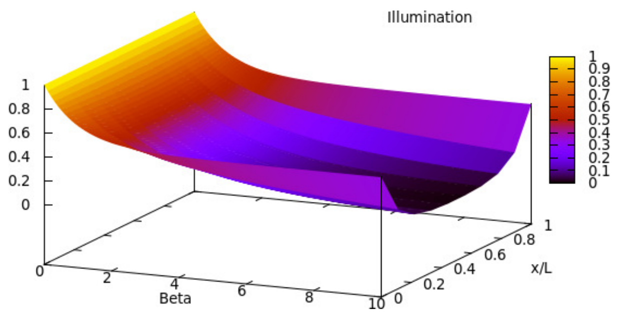


FIGURE 7.13: Plot of illumination function with $\gamma = 1$ and without mutual cell shading effects.

The final process developed for the photobioreactor model is shown in Algorithm 17. Notable differences with Algorithms 16 and 17 include the 3x3x3 block sizes which are also used with a Monte Carlo style update scheme as described earlier. In addition, a cell death probability is introduced, alongside cell shading. Parameters used in experiments for this final algorithm are shown in Table 7.1.

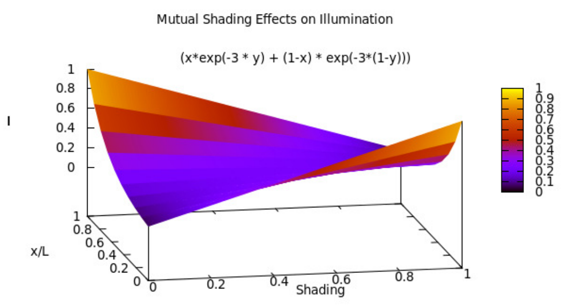


FIGURE 7.14: Mutual shading effects on illumination, assuming shading symmetry (ie. $S_L = L - S_R$): $I = (xe^{-\beta y} + (1-x)e^{-\beta(1-y)})$. The two peaks in this plot represent the left and right edges of the cultivation vessel.

```
$N = W * H * D$ for 3D lattice
blank lattice, initialise 10 random cells
for all time-steps do
    compute cell shading map, CUDA
    for all 3x3x3 blocks in lattice do
        for all $3^3$ sites $i$ in each block, random order do
            thread return if out of bounds
            thread return if empty cell

            compute left and right shading (scalar values, [0,1])
            compute illumination on site, $I$
            kill cell and return on probability cell_death_prob $* I$.
            choose random neighbour site
            perform state transitions and state imperatives
            perform Kawasaki site exchange
        end for
    end for
end for
```

ALGORITHM 17: Kawasaki site exchange algorithm including the PSF model and mutual shading effects.

Parameter Name	**Value**	**Description**
Temperature	0.4	Constant temperature for metropolis probability calculations.
Gravity	0.1	A weak gravitational bias on the metropolis probability.
Growth Rate Scale	0.3	This value is used to scale the aggregate transition counts from activated to resting, in order to determine cell division probability.
Illumination γ	0.5	Scaling of illumination I, where $0 \leq I \leq 1$.
Illumination β	5.0	Exponent of photo attenuation into the medium. Higher values attenuate light faster, and cause less light to penetrate toward the centre of the vessel.
α	0.8	Probability that a cell transitions from **resting** state to **activated** state.
β	0.7	Probability that a cell transitions from **activated** state to **inhibited** state.
γ	0.1	Probability that a cell transitions from **activated** state to **resting** state.
δ	0.01	Probability that a cell transitions from **inhibited** state to **resting** state.
Cell Death Probability	0.001	Probability coefficient applied to I and $1 - I$ to determine whether to destroy a cell (Used for including biomass reduction at over- and under-illumination.)

TABLE 7.1: Parameter settings and their descriptions. Values are held constant unless noted otherwise.

It is prudent to point out that it is unrealistic to assume that all agent-based systems can be modelled as elegantly as “Boids” [240]. Models with excessive numbers of parameters are limited in their prediction capabilities to a certain extent, however. For instance, Eilers and Peeters conceded that their updated model of 1993 [58] with six parameters was a large number, presenting possible difficulty in fitting to collected data. Nevertheless, the parameters in Table 7.1 are hand-calibrated.

The first observation made was that while overilluminating the vessel, resulting in photoinhibition of the cells, cells that spontaneously form clusters are able to grow (albeit slowly) and thrive, due to mutual shading reducing illumination to more acceptable levels. An example of this is shown in Fig. 7.15. The CUDA kernel computing the shading map proved quite computationally expensive however, at around 3.3ms.

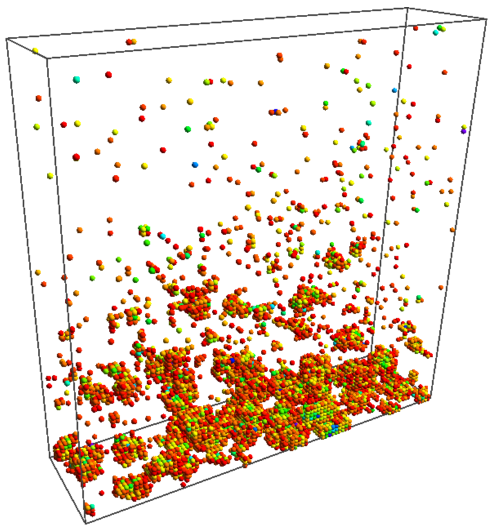


FIGURE 7.15: Example of how dense cultures thrive under photoinhibition due to cell shading.

Shown in Figure 7.16 is the vessel fill ratio with illumination conditions varying at timesteps 0 ($I = 1.0$), 10^4 ($I = 0.01$) and $2 \cdot 10^4$ ($I = 1.0$). The purpose of this was to see the effects of underillumination. What is interesting to note, is that while there is an immediate decline at $t = 10000$, there is a slow increase towards $t = 20000$, during which illumination was still prohibitively low. Observations would suggest that this is caused by cell deaths decreasing mutual shading, and hence allowing more light towards the centre of the vessel.

Another observation made is that various values of the light attenuation exponent (β) resulted in a different spatial configuration of the culture. Lower values of β (very clear liquid) saw the culture thriving in the centre of the vessel, due to overillumination at the edges. Higher values of β (cloudy liquid) saw the culture adhering to the left and right walls.

The image shown in Figure 7.17 is rendered using isosurface extraction with Phong shading and a marching cubes algorithm implementation in a modified CUDA SDK sample "Marching Cubes" [202]. An appreciation of depth is much easier in this image. Clusters on the edge of the vessel are not closed, however.

Cursory performance indications were concerning, since for every random site chosen in the 3x3x3 sub-lattice, one CUDA kernel must be executed synchronously across the entire lattice. A total of 3^3 CUDA kernels[5] are computed for one timestep (cubes of dimensions 3x3x3). Moreover, each of these individual kernels, while fast for a relatively sparse vessel, become more cumbersome as the vessel becomes more dense. While the vessel has a $< 1\%$ fill ratio, one such kernel executes for a duration of approximately $93\mu s$, including synchronisation, which accounts for around $2/3$ of this. At approximately 80% fill, this increases to around $300\mu s$, for each of the 27 kernels. This equates to little more than twice the shading map kernel. Given that mutual cell shading is not giving much more realism than an appropriate global illumination attenuance, this result would suggest that computing cell shading manually is simply too computationally expensive, moreover, system size scaling would be heavily affected.

Having arrived at a bioreactor model which takes into account mutual cell shading, negative biomass growth, illumination, cell division, approximate hydrodynamics and parallelisation, the next stage in this chapter is to consider the use of the MOL language in terms of utility and performance, and examine the differences by comparison.

7.3 Modelling using MOL

($\mathcal{I}$, p. 107)

MOL is a high-level, extensible language previously proposed and discussed in Chapters 5 and 6 which compiles at runtime to machine code. It uses the run-time program compilation features of Terra, indirectly facilitated by LLVM (backends including X86 and PTX) and supported by the high-level nature of Lua.

Certainly of immediate interest in MOL is automatic parallelism. Once a model is written, it can be compiled for CPU or GPU architectures without additional code. Extensions to MOL do have to take this into account, but future versions will eliminate this requirement. The objective of this section is to discuss the complete implementation of the agent-based photobioreactor model presented in section 7.2 in MOL. The MOL model is then quantified and performance characteristics are analysed. Following this, some structural optimisation experiments are carried out. The results of this are detailed and discussed.

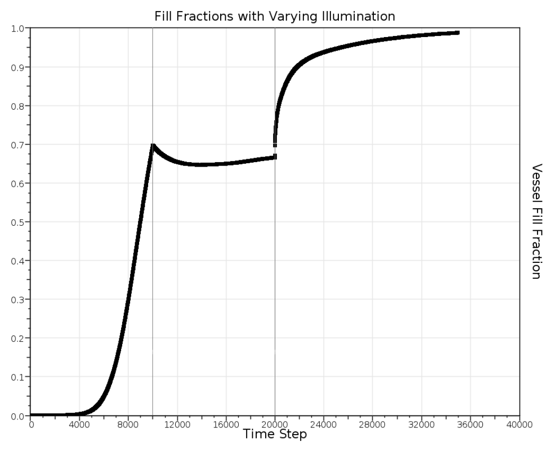


FIGURE 7.16: Sample photobioreactor fill fractions for varying parameters at selected intervals.

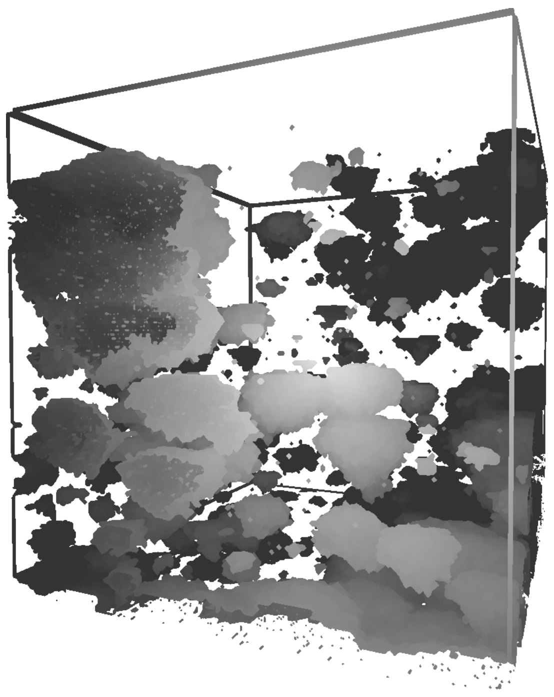


FIGURE 7.17: An example of isosurface rendering of a partially occupied photobioreactor.

```
mol
  if me > 0 then

    defvar I = compute_illuminance
    defvar state = get_my_state
    defvar myspecies = get_my_species
    defvar activated_counter
      = get_activated_state_counter

    if state == ACTIVATED then
      if (randomfloat) < GAMMA then
        increment_activated_state_counter
        go_to_resting
      else
        if randomfloat < BETA * I then
          go_to_inhibited
        end
      end
    else
      if state == INHIBITED then
        if randomfloat < DELTA then
          go_to_resting
        end
      else
        if state == RESTING then

          if randomfloat
            < ( get_split_probability ) then
            select all
              make_split
            end
          else
            if randomfloat < (ALPHA*I) then
              go_to_activated
            end
          end

        end

      end
    end

    defvar dir = get_kawasaki_move

    if dir ~= −1 then
      move dir
    end
    d = 1
  end
end
```

LISTING 7.1: Photobioreactor model from Section 7.2 reimplemented in MOL.

Extension	Description
compute_illuminance	Calculates the influence of left and right illumination sources, and takes into account mutual cell shading from either side of the reactor vessel.
compute_left_shading	Calculates the cell shading effect on illumination from the **left** of the reactor vessel.
compute_right_shading	Calculates the cell shading effect on illumination from the **right** of the reactor vessel.
get_my_species	Computes the species of the current cell.
get_my_state	Computes the state the current cell is in.
go_to_activated	Transitions the cell to the **activated** state.
go_to_resting	Transitions the cell to the **resting** state.
go_to_inhibited	Transitions the cell to the **inhibited** state.
get_split_probability	Computes the probability that a cell will divide.
count_current_like_like_bonds	Counts the current like-like (LL) bonds for the current cell by examining nearest neighbours (NN).
get_kawasaki_move	Computes energy and Metropolis probability, returns a direction if the exchange is accepted.
increment_activated_state_counter	Increments the cell's activated state counter.
make_split	Creates the code necessary to cause cell division to take place.

TABLE 7.2: MOL extensions written to incorporate bioreactor states and transitions.

($\mathcal{I}$, p. 117)

7.3.1 Implementation Detail

The MOL code for the photobioreactor model is shown in Listing 7.1. A table of the MOL extensions used in Listing 7.1 is provided in Table 7.2. For some of the extensions listed, bitwise operations are implemented in C, which is compiled by Clang into LLVM IR. This means that ordinary C code is also able to be compiled to CUDA PTX alongside Terra (and therefore, MOL).

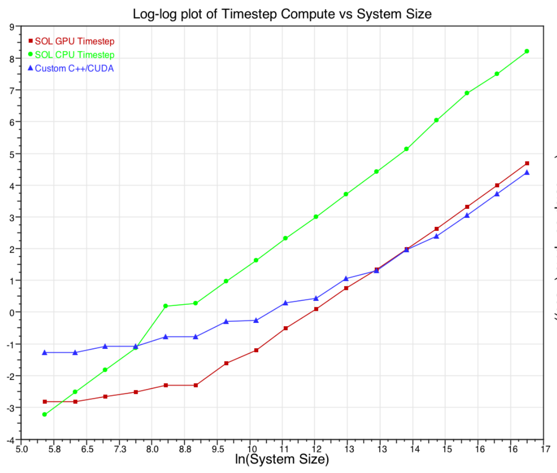
Some variables and keywords previously unused in MOL are put to use in this program. The keyword randomfloat computes a random floating point number in the range $[0, 1]$ using the Ran random number generator [229]. In addition to having one Ran and its internal state, when compiled for GPU, each executing thread has its own Ran and internal state initialised at the start of the program. This allows the algorithm to use as many random deviates as necessary without additional storage.

It is certainly notable that these extensions can be placed in a library. It is envisioned that this will be done in future versions of MOL. At this point in time, the user is able to write the model with parts in MOL, Terra, and C/C++, depending on how extensive the libraries of extensions are. Certainly providing easy access to more powerful languages aside from MOL makes it much easier for experts to use.

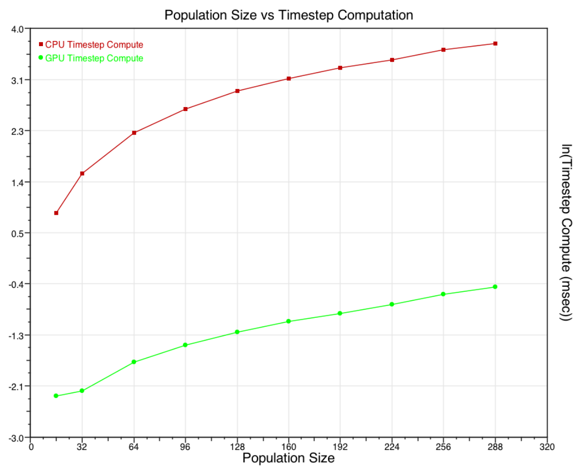
This implementation focusses on a 2-dimensional implementation, as the 3-dimensional version of the model discussed above did not contribute as much to the growth rates as was expected.

In the next section some system size scaling experiments are performed for various parallel configurations, and this is followed by a structural optimisation experiment.

7.3.2 Results



(a) Log-log plot of the time taken to compute a single timestep across a range of system sizes for the **B1M* MOL parallelisation strategy for CUDA, the X86 compile of the identical MOL code, and finally the custom-written C++/CUDA code discussed in Section 7.2.



(b) Log-linear plot of timestep computation time (in log_e scale) against population size of *heterogeneous* programs.

FIGURE 7.18: Performance results of MOL, including comparison with custom C++ code.

[5]Excluding the mutual cell shading map kernel.

Performance results collected are compared against that of the custom written C++/CUDA code given in Section 7.2. Figure 7.18(a) shows that for the smallest system size (16x16), the X86 compiled version of the MOL code is superior. This is expected, considering that the memory copies involved for the use of GPU hardware are very costly. The CUDA compiled version of the MOL code quickly surpasses the single-threaded code, and also keeps an edge in computing time over the custom written code previously discussed until much larger system sizes are reached. This change in performance is likely due to minor algorithmic differences in the two implementations, most notably the computation of mutual cell shading. In the custom written C++/CUDA code, a "shading map" is computed using a separate kernel, which has a CUDA grid with a single yz plane, which sweeps through the rest of the lattice, first from left-to-right, then right-to-left. The MOL version uses the same kernel for computing these values, which results in redundant computations. For the next section on structural optimisation, extremely large systems sizes are less important, however.

In both Figures 7.18(b) and 7.18(a) the coarse checkerboard update was disabled to reduce algorithmic differences as much as possible.

Figure 7.18(b) is a log-linear plot of time taken to compute a single timestep, against the size of the population. The population refers to the number of candidate models of size 32x32. Each data point is averaged across the first 60 frames of a randomly initialised set of candidate models. The data was collected for a *heterogeneous* population of candidates. This was accomplished by adding an arbitrary `select recombination` statement around the state transition code in Listing 7.1. The objective function is unimportant, as the goal was to produce a heterogeneous population and determine how this scales with respect to the number of candidates. As shown, the CPU timestep compute times increase much faster than the GPU counterpart. A population size of 256 is very suitable for an evolutionary algorithm, which is tolerable for the GPU code, but not for CPU. Population sizes smaller than 32 seem to lack in diversity and have generally less success in optimisation than when population sizes are 64 or greater.

7.3.3 Structural Optimisation Experiment (*I*, p. 128)

The task of evolving a very simple finite state machine to accomplish an arbitrary goal will be examined. Suppose a modeller wanted to determine if a state machine of similar complexity to that shown in Listing 7.1 could produce a large number of cells and absorb as much illumination as possible, given the effects of mutual cell shading and light penetration. A very simplistic objective function could be the sum of the total number of cells and total illumination units absorbed. The modifications to the code is shown in Listing 7.2.

Another important modification added to the code is on line 33. The reason for wrapping the crucial cell division code segment in a `select all` block (disallowing the optimiser to deconstruct and recombine within it) is to impose the importance of $P(\text{split})$ on the actual cell division operation. Should the optimiser be able to use only the cell division extension (`make_split`) then an overwhelming number of cells will be created. Effectively, this accomplishes a certain search-space limit. Any modification may be made, except within the `select all` block.

The fitness function is somewhat peculiar in this arrangement. At this point, the fitness function is evaluated for every cell, in order to cover the broadest possible range of fitness function specifications. This means that the expression `total + I/1000` will be evaluated once every timestep, for every live cell and added to the score of the candidate model. The `total` value, however, will be evaluated only on timestep `1000`, but it will be executed by all live cells. The score given is therefore dependent on the number of live cells. It is for this reason that `total` was moved to lines 3–6, in order to ensure that only one cell does this computation, and only on the final timestep. This is the purpose of the `timestep if` statement. The `I/1000` term of the fitness function is the computed illuminance for all cells, averaged over 1000 time steps.

```
mol
  defvar d = 0
  defvar total = 0
  if timestep == 1000 then
    total = count_live_cells / (30*30)
  end
  if me > 0 then

    defvar I = compute_illuminance
    defvar state = get_my_state
    defvar myspecies = get_my_species
    defvar activated_counter
      = get_activated_state_counter

    select recombination
      to maximise(total+I/1000)
--[[I0]]  if state == ACTIVATED then
--[[I1]]    if (randomfloat) < GAMMA then
--[[N0]]    increment_activated_state_count
--[[N1]]      go_to_resting
        else
--[[I2]]      if randomfloat < BETA * I then
--[[N2]]        go_to_inhibited
          end
        end
      else
--[[I3]]    if state == INHIBITED then
--[[I4]]      if randomfloat < DELTA then
--[[N3]]        go_to_resting
          end
        else
--[[I5]]      if state == RESTING then
--[[L0]]        select all
              if randomfloat
                < ( get_split_probability ) then
                select all
                  make_split
                end
              else
                if randomfloat < (ALPHA*I) then
                  go_to_activated
                end
              end
            end

          end

        end
      end
    end

    if d == 0 then
      defvar dir = get_kawasaki_move

      if dir ~= -1 then
        move dir
      end
      d = 1
    end
  end
end
```

LISTING 7.2: Photobioreactor model from Section 7.2 with a recombination structure designed to search for candidates which maximise cell count and illumination.

The lines in the code which are marked with a comment in the form of a letter and a digit are lines which are stored as functions and terminals in the database of the optimiser. Those which start with ‘I’ are 2-arity functions, those which start with ‘L’ are statement lists which cannot be broken down, and ‘N’ denotes anonymous expressions, which are in fact MOL extensions in this case.

To execute this test, parameters from Table 7.1 are used. Lattice sizes were set to 30x30 to facilitate the coarse checkerboard update ensuring that no race conditions take place. Finally, the number of candidates in the population was set to 64; which ensures that code compilation is not completely overwhelming.

In terms of performance, some rough indications are given here. To evaluate a single generation of candidate models, 1000 timesteps were computed, and repeated 16 times to obtain an average. One timestep lasted approximately 0.38msec, and CUDA kernel compile times lasted approximately 26.8s, and 28.7s for the entire generation compilation from MOL to machine code. MOL code was compiled to Terra ranging between 7msec and 315msec, the average being about 15msec.

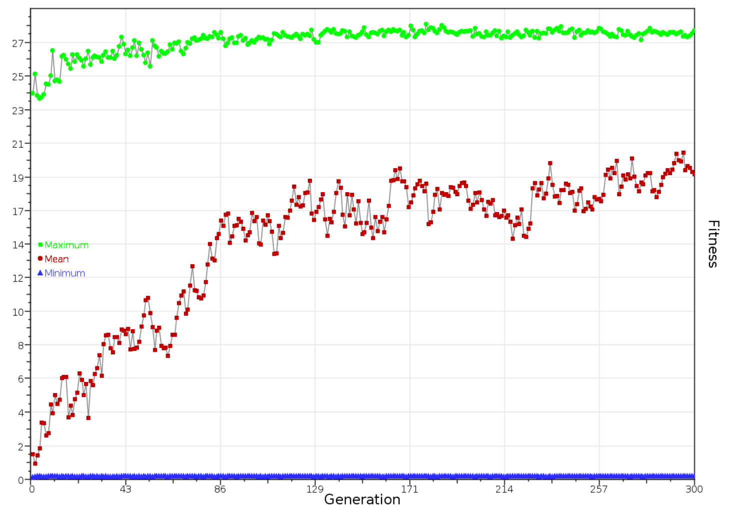


FIGURE 7.19: Fitness plot of the recombination of state transition code over 300 runs.

Following the recommendations of Ferreira [63], GP-style elitism was used to bias the search more towards fitter programs. This was done by selecting the best program in a given generation, and then duplicating this program into randomised locations in the programs array. For the purposes of this experiment, three such copies are made per generation. This follows that generations will never have a decrease in maximum fitness. However, due to the fact that several functions and terminals are stochastic, several successive evaluations may differ slightly. To combat this, results were averaged 16 times as discussed above. However, some small differences may still present themselves, as is evident in the fitness plot shown in Figure 7.19.

In Figure 7.19 the mean fitness increases steadily. A slight increase in maximum fitness is present until around generation 100. Candidates steadily increase in fitness, occasionally discovering a model program which performs better than the previous ones. The maximum fitness data points are created by the most “fit” programs generated. One of these programs were taken and is analysed below.

Figure 7.20 a visual representation of the *k*-expression:

P0P0I2L0I1P1L0I2P0P0I5N1P1N2P1I5N3L0L0N1N1N0N1L0N3N0N2N0

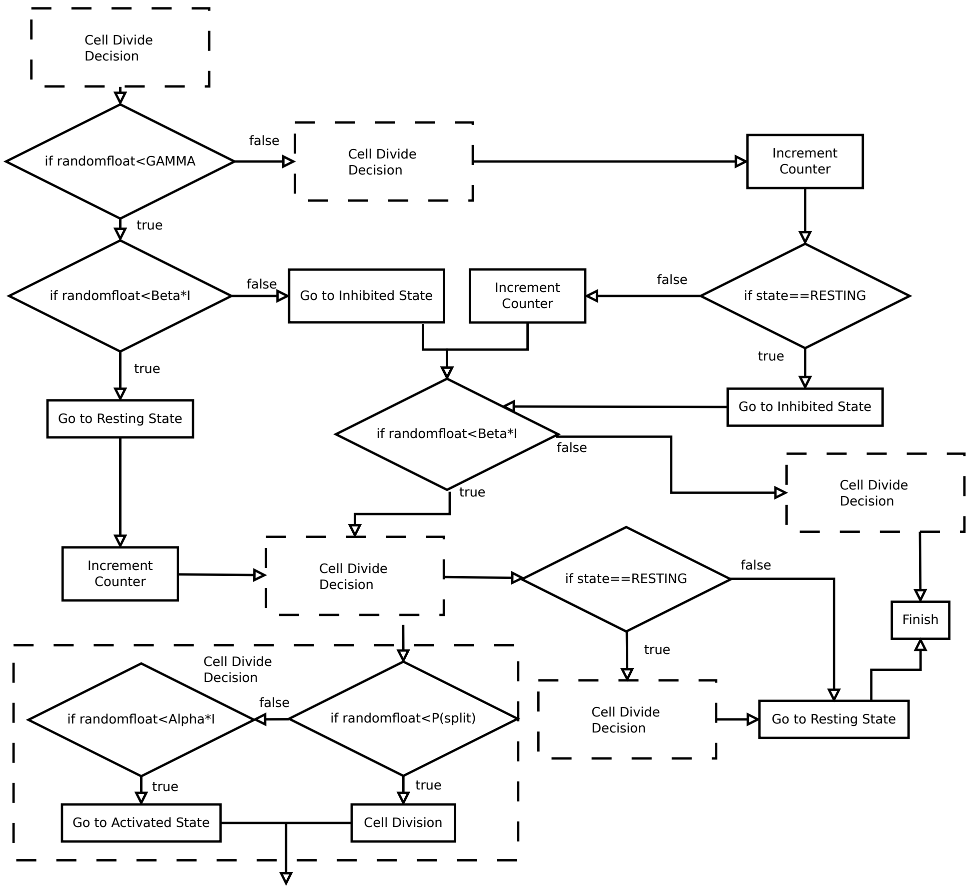


FIGURE 7.20: Candidate model program with redundancies removed, as a flow diagram.

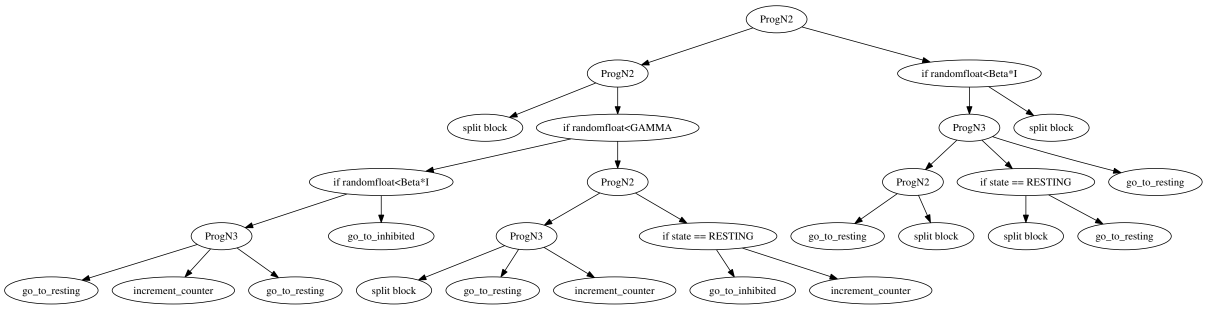


FIGURE 7.21: Candidate model program in the form of an abstract syntax tree.

The figure is in the form of a flow diagram, constructed from the abstract syntax tree of the above expression shown in Figure 7.21. The expression above excludes 25 *introns* of the genotype, which have no effect on the phenotypic interpretation shown in Figure 7.21. The cell division section is shown at the bottom left.

It is immediately clear that there are several redundancies in the program shown in Figure 7.21, which is not unlike that experienced by Koza in the Santa Fe Ant Trail problem [149]. Figure 7.20 shows a flow diagram representing the actual flow of the program without redundancies. Though some semantic meaning is lost from the original intention of the nonterminals and terminals provided, the split probability restriction (facilitated by the select all block) forced the optimiser to search for a program which increases $P(\text{split})$ as well as increasing the amount of illumination absorbed. Since $P(\text{split})$ depends on illumination and the activated state counter, the optimal program is less intuitively created. Hence, it is appropriate to use an optimiser to find a program which may satisfy this requirement. The reason why there were many redundancies in the program is because the objective function does not favour simpler programs.

It is interesting to note from the flow diagram, that it still contains a number of state transitions. The multiple cell division attempts is simply due to the fact that it increases the probability of a split occurring, without affecting $P(\text{split})$ which it cannot.

It is difficult to interpret the flow diagram, but from visual inspection of the simulation, it appeared that cell growth was fast, and the counters were increased quickly, except for cells in relatively high lighting conditions.

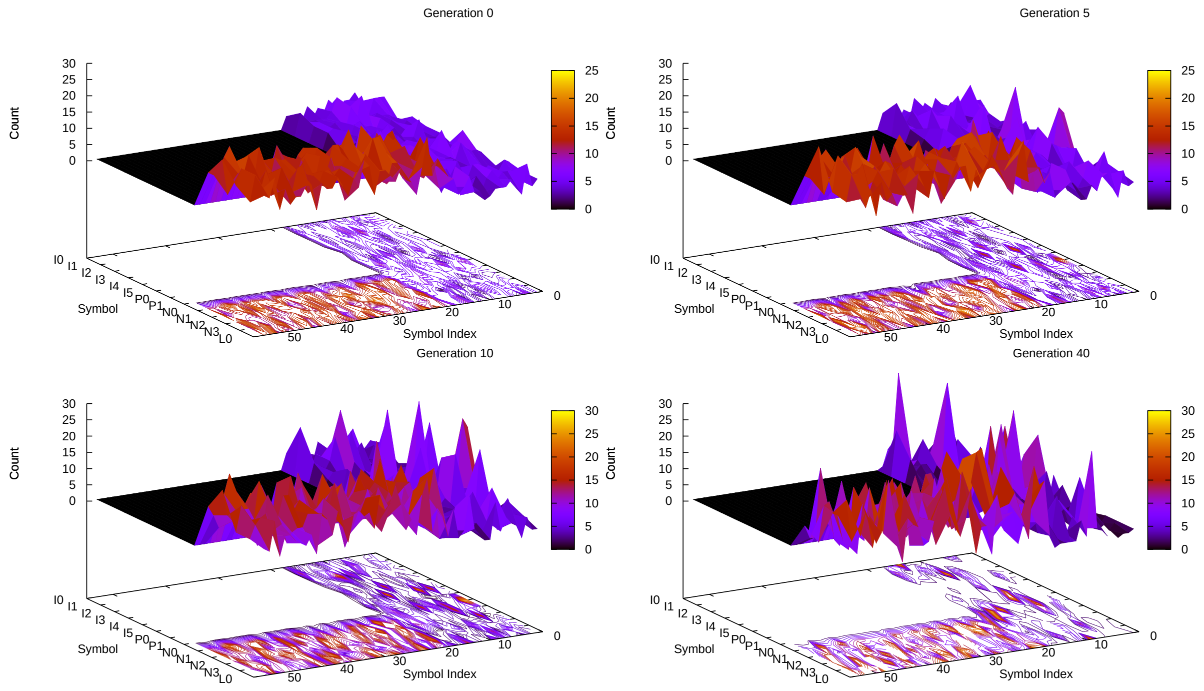


FIGURE 7.22: Symbol histograms of k-expressions for generations 0, 5, 10, and 40 in a sample run.

Figures 7.22 gives two symbol histograms indicating the frequency that various symbols occur in specific indices in the k-expressions of four generations (0, 5, 10 and 40). The dark black region exists because no 2- or 3-arity functions may be present in the tail-section of a k-expression. Therefore, a higher agreement is shown for terminal symbols and indices 17–53. Indices 0 to 17 may contain any symbol. Generation 0 is a freshly initialised set of candidates, and it is therefore unsurprising that they do not agree on symbols and their placements.

What is interesting to note on these graphs is that symbol and index agreements surface quite quickly, and may reach as high as 30 for certain symbols in only 40 generations. The contour lines provide an easier to interpret representation. The fourth image shows high numbers of candidates which have the `I0` or `P0` nonterminals at index 0. Candidates with a terminal symbol at index 0 earn very low scores. The `N0` terminal (which increments a counter) seems to be very highly used in generation 40, as opposed to the other generations. In summary, these graphs usefully show that over time, candidates begin to agree over their choices of symbols while still preserving the crucial diversity for the optimiser to function.

7.4 Discussion

In summary, a concerted effort was made in first proposing an agent-based model of a photobioreactor. This model was adapted three times to include the effects of cell division, mutual cell shading, negative biomass growth, and approximate hydrodynamic flow. This model was implemented in C++ with CUDA. Finally the model was quantified and performance measurements were gathered. The model was then reimplemented in MOL, a language proposed and discussed in the previous two chapters. The two models were then compared qualitatively and performance-wise.

The purpose of the chapter was to demonstrate the utility of MOL in modelling, performance and selective model induction. MOL appeared to suit modelling well, though it must be admitted that a great many extensions had to be written to implement the model (see Table 7.2). Though these extensions can now be reused in other models, they are very model-specific. It is possible to use Terra functions in global scope directly in MOL programs, which could also simplify the code. In this case, to keep the model code as simple as possible, only extensions were used.

The model optimisation experiment conducted was successful in generating programs which satisfied the objective function to some extent. Criticisms of genetic programming and evolutionary algorithms in general certainly still apply in this situation. Most notably, the considerable problem of specifying the actual objective function is placed firmly on the user. One significant issue of specifying objective functions is to ensure that the search space is sufficiently small. This is made somewhat easier by allowing the user to restrict the search space in a variety of ways, and to observe the result qualitatively while the optimiser is running. Statements such as `select all` and timestep-specific objective function computation were more useful than previously anticipated.

In summary, the MOL code allowed the model to be described in a fashion which was easy to read. The code compiles to machine code through Terra and LLVM, allowing it to be optimised. The ability of MOL code to be compiled as single-threaded or GPU-parallel without additional modifications also makes it very useful in cases where larger systems are necessary. In this case, GPU-parallelism was used to improve optimisation performance. The built-in optimisation algorithm also produced interesting results, which would otherwise require extensive modifications to the C++ code.

There remains future scope in multiple areas of this research. Improvements in syntax, extensions of libraries, user interface design, other optimisation algorithms and debug tools would help tremendously to bring this research in ready-to-use form to the end-user.

Part IV

Discussions and Conclusions

CHAPTER 8 DISCUSSIONS

Major areas covered in this dissertation include parallel agent-based modelling and simulation, parallel continuous optimisation, parallel combinatorial optimisation, a new domain-specific language for ABM and the addition of an optimiser in this language. Finally, a thorough study was done on the use of this language and its features, in comparison to a more conventional approach using custom written parallel C++ code. The major contributions of the dissertation are discussed here, and conclusions are made in Chapter 9.

8.1 Parallel Agent-based Modelling and Simulation

IT HAS BEEN DEMONSTRATED (in Chapter 2) that the inherent parallel nature of large-scale agent-based models is exploited with relative ease across graphics processing hardware. The increased performance brings with it access to larger systems, which are gaining increasing interest in recent years. For instance, biological systems such as algal photobioreactors can be modelled only at very small scale at present. Graphics processing hardware certainly show great promise to accelerate such simulations on the laboratory desktop computer. The use of Graphical Processing Units (GPUs) in some of the world's fastest supercomputers further attest to this.

Although there appears to be good scaling characteristics of models such as Boids across GPUs, a significant issue remains. Computational complexity of models such as Boids cause a very fast increase in timestep computing time as system sizes increase. This complexity arises from the interaction between agents. Where an agent must determine whether another agent is within communication distance, it must compute a distance and test it for every agent. The vast majority of authors in the Agent-based Modelling (ABM) community agree that agent-agent interaction is a crucially important aspect of ABM.

Fortunately, this problem has been considered in the N-body literature [203] as early as 1985 by Andrew Appel [3] since before the wide acceptance of ABM. Two major schools of thought exist in the reduction of complexity in systems such as these: approximations such as the Barnes-Hut method [12], and redundancy elimination. The former deals with approximating the collective effect of a cluster of agents (such as that of Scheffer et al. [252], though, for a different purpose), the latter deals with the elimination of redundant interactions. Should an agent only communicate and interact with agents in close proximity, then any method which reduces the number of distance computations which do not result in interaction, should be employed. A good method of accomplishing this is by using spatial partitioning techniques, which uses fast sorting algorithms to group together agents with a guarantee that all agents within range will still interact.

While spatial partitioning techniques acceptably accelerate simulations in single-threaded environments, the use of these on a data-parallel architecture is less trivial. Many spatial partitioning algorithms construct their datastructures by recursion using pointer trees. Although recent advances in graphics hardware have improved support for recursion and dynamic memory allocation, these are still less trivial to utilise effectively. Warp divergence caused by branching code also reduces throughput due to the fact that the SIMT architecture forces threads to execute the same instructions, meaning that threads which diverge are simply disabled (executing NOOPs) until their execution paths align again. For these reasons, a number of spatial partitioning algorithms have surfaced for graphics hardware.

Chapter 2 discussed a spatial partitioning technique commonly termed the uniform grid [84] in the context of ABM. Green [84] used the algorithm and datastructure to accelerate collision detection computations in a particle simulation accelerated by NVIDIA's CUDA. Considerable performance improvements are obtainable from using this technique. Results from testing this algorithm in the context of ABM reiterated this, but presented a shortfall. Assuming that particles are uniformly distributed across the space (a condition always satisfied in the simulation of Green), then an acceptably smaller number of adjacent particles will be iterated over for each particle during timestep computations. However, in the case of Boids, where collision detection is less important (more flocking behaviours), agents can contract around a specific location, causing complexity increases beyond $\mathcal{O}(n^2)$ due to the fact that all agents must again communicate. Not only this, but a datastructure must still be computed which (depending on the algorithm) sometimes reaches $\mathcal{O}(n^2)$ itself.

In summary, it seems that spatial partitioning can be useful, but the appropriate algorithm must always be chosen. The multi-stage programming paradigm which features in this dissertation in Chapters 5 and 6 provide the ability to "store" such algorithms and their datastructures for later use. The addition of compile-time checks may also yield the appropriate choice of algorithm, alongside "hints" from the user and run-time information collection.

With the improved performance afforded by using a single graphics processing unit, it is not unreasonable to consider the use of several. Section 2.2.3 considers this along with the use of a uniform grid spatial partitioning algorithm. Given that algorithmic improvements could be made, it was encouraging to see that the multiple-GPU version of the model was able to be supported by the same spatial partitioning algorithm and datastructure. The result was a reasonably scalable simulation, bounded by host memory. With additional improvements, storing the simulation entirely upon graphics hardware would see this restriction removed and a model could then be arbitrarily scalable, bounded only by the number of graphics processors on a single machine. The next step would be to use multiple host nodes in a high performance cluster with multiple graphics processors each.

Throughout the discussion of these techniques in Chapter 2, visualisation took a secondary role consistently providing qualitative feedback, complemented by quantitative methods including techniques such as clustering and spatial histogramming. Visualisation methods are very important and deserve developmental efforts proportional to that of the model itself. Building a model without qualitative feedback is a difficult and error-prone endeavour.

Visualisation also presents its own problems, particularly that of performance. It is for this reason that performance measurements were always taken with no visualisation. Rendering a set of agents upwards of one million is particularly time consuming. For these simulations, pixel shaders were used to render agents as spheres. The alternative was to render Boids, for example, as agents indicating their current heading. While helpful for debugging purposes, rendering slows tremendously. Concepts similar to laboratory-bench scale experiments in building large-scale agent-based models seem more and more appropriate. A technique not taken into account in Chapter2 was the use of pixel buffer objects (PBOs), which take advantage of the fact that computation of data destined to be rendered take place on the graphics processing unit itself. While this removes some of the need to constantly copy memory between host and device (a computationally costly process), it would still be necessary to endure the waits for rendering many triangles, which could be mitigated by the same point-rendering methods already mentioned.

8.2 Parallel Continuous Optimisation

The similarities between agent-based models and continuous-space optimisers such as the Particle Swarm Optimiser (PSO) can perhaps best be demonstrated by the time it takes to convert one to the other. There is a special relationship between the PSO and the Boids model. As the authors of the PSO note [142], it was inspired by flocking and schooling behaviour of birds and fish. This interrelationship enables one to apply techniques for performance from one on the other.

Introductions to global continuous optimisation algorithms were given in Chapter 3, with a particular focus on population-based evolutionary algorithms. Like agent-based models, these algorithms offer a natural description which can easily be related to analogous processes in biological evolution. In addition, they also have an inherent parallel nature, which can be effectively exploited.

The work of Wolpert and Macready [294] provides proofs (in the form of "No Free Lunch" theorems) with far-reaching implications on the global optimisation literature. Their work formalises the notion that choosing an algorithm for a particular optimisation problem is indeed subject to error. Effectively their theorems infer that there is no algorithm which performs better than all others across all problem domains.

Therefore, efforts towards implementing new algorithms continue in earnest. Section 3.3 considered the use of advanced space exploration techniques to improve the search for the global optimum. These were tested across a range of test functions, eventually leading to (albeit limited) convergence performance increase. Although there is some disagreement in the literature on the validity of Lévy flights in some areas, it appears that such a searching tactic does indeed aid the exploration of the search space. Lévy flights should be used with caution however, as they are computationally expensive to generate. Although they can be approximated [300], they may not always be an improvement upon the usual random walk, as demonstrated by the Rastrigin function in Section 3.3.1.

Two global optimisation algorithms and their parallelisations were discussed in Chapter 3: the parallel *Many-optimising-liaisons* PSO (MOLPSO) and the Firefly Algorithm (FA). It was found that saturating the search space with searching particles for either the FA or MOLPSO would generally improve the quality of the solution found. This was especially the case with the Schwefel function, which is a test function traditionally tackled with large bounds. The same could not be said with certainty for the Rosenbrock function. The variable dependence in this function and large neurtal valley makes searching very difficult, even for a large number of particles. Regardless, the performance increase attained using graphics processing hardware with Lévy flights does improve the chances of finding a good solution dramatically, and in much less time than single-threaded algorithms.

Even the simplest global optimisation algorithms require some form of calibration. The MOLPSO, a simplified PSO still requires two parameters to be specified. The FA is no different. Depending on these parameter settings, algorithms can behave radically different. This often makes the difference between finding an optimum, and succumbing to a local optimum. Recommended settings may suit some problems, but not all. In an effort to mitigate this, meta-optimisation was examined. Meta-optimisation seeks to use an optimiser on another optimiser for parameter calibration. This was considered in Section 3.5.

A concern of using a meta-optimisation process to pre-calibrate an optimiser is the loss of generality involved. As discovered in the experiments conducted, over-fitting can become a problem. Should parameters be optimised with respect to a specific function such as the Rastrigin function, then that optimiser will tend to be less effective on other test functions. This is not unlike similar problems encountered in the training of neural networks, where it is now widely recognised that separate sample sets are necessary for reducing the chance of over-fitting.

Another considerable issue is the computational complexity involved in the process of using an optimiser on itself. Particularly when both optimisers involve populations of candidates. For this reason, the meta-optimisation algorithm discussed in Section 3.5 was again parallelised across graphics hardware. It is perhaps the same reason why meta-optimisation has received relatively little interest. In summary, experimenting with optimisers in this sense has given some useful feedback on calibrating agent-based models, given that the PSO is in principle, directly related to flocking agent-based models.

Having discussed optimisers which search through the n-dimensional hyperspace of certain test functions, it is certainly also useful to gain a *qualitative* sense of how these operate in order to improve them. It is worthwhile to inspect an optimiser on its ability to optimise 2-parameter or even 3-parameter functions, which are trivial to visualise. Higher dimensions are less trivial, however, but quite common. A simple technique was developed to visualise particles in higher dimensions. The overly simple method used had at least resulted in the ability to visually identify the optimum from distinctive features.

8.3 Parallel Combinatorial Optimisation

The use of parallel hardware such as graphics processing units on combinatorial optimisation problems are not straightforward, and arguably less so than global continuous optimisation problems. This is especially true for problems which require the search for a suitable program. The first problem one might encounter is the choice in representation. Linear representations do exist, and are most suitable for graphics hardware, as they can be stored in memory without the use of pointer datastructures.

The reason for considering geometric optimisation in a less formal manner in Chapter 4 was to facilitate some cross-discipline synergy between continuous global optimisers and combinatorial optimisers. By bringing the Firefly Algorithm from continuous optimisation to program-space optimisation, an interesting effect was that the fitness decay seemed to improve convergence performance, as it did in continuous space. Moreover, decay parameters have a dramatic effect on the outcome of the optimiser, more so than anticipated. While an improvement, it does introduce one additional parameter that may require problem-dependent calibration, which is undesirable.

The linear representation of candidate programs named *Karva* proved to be particularly useful. It more closely aligns to the natural phenomena which inspired Genetic Algorithms and Genetic Programming. The concept of intron symbols in a program provides a simple and elegant method for storing disabled symbols, which can later be potentially reactivated. Using this representation on graphics processing units is very successful, even including the genetic operators themselves, but some special considerations are necessary to maintain the structure of these and traverse them correctly.

Although implementing genetic operators to operate on Karva expressions is relatively simple, the traversal of these trees is less trivial. The trees must be interpreted in a somewhat awkward manner, which prompted the development of an additional CUDA kernel to be executed simply to map arguments to their functions. This tradeoff with a loss of some performance was beneficial, in order to gain the elegant advantages of Karva such as intron symbols.

It should be noted that the algorithm for which the Karva language was developed, Gene Expression Programming (GEP), is indeed much more complex. It contains several genetic operators in addition to simple mutation and crossover which ensure that effective circulation of genetic information takes place [63]. It is therefore appropriate to consider the entire set of GEP operators, and not only simple point mutation and one-point crossover. Even though these were not considered, good performance was still attained. This area is promising for future research.

Another algorithm discussed in the context of Karva and GPUs is *K-GPSO-GPU*. This modified version of the PSO was made using the geometric paradigm, in order to adapt the PSO for program-space search. It was compared to a parallel Genetic Programming algorithm also operating on Karva-expressions (the *K-GP-GPU* algorithm). Interestingly, results suggest that the K-GP-GPU algorithm was more suited to solving the problem. Further, it would appear as though the K-GPSO-GPU algorithm lacked the necessary population diversity to maintain a successful search. While the algorithm did not fare well on the modified 3-dimensional Santa Fe Ant Trail problem, it may still be possible that this algorithm is more suited to a different problem domain.

Further to the concept of geometric optimisation, it is important to consider appropriate distance metrics. Measuring the distance between two candidates by Hamming distance was accurate but somewhat arbitrary. The optimisers could benefit from a different measure, such as the various edit distances [22]. Once again, this may be subject to problem-domain variation.

Finally, program-space visualisation was considered in the context of the algorithms presented in Chapter 4. Two techniques were presented. One involved the use of a graph rendering package to render a population in the form of a tree, where nodes are shared among candidates, and each level of the tree represents the next symbol index in the Karva expression. The other technique was perhaps more simple, and involved the recursive division of space, which could be inspected by scaling and translating the viewport. Two different methods were used to compensate for their relative weaknesses. The recursive method suffers from a fast loss of its comparative qualities with larger Karva expressions, and the graph-based method lacks the ability to describe the specific symbols used without viewing the image in large scale. Together, these methods complement each other well, are simple to implement, and provide useful information which may assist the designer of a program-space (or indeed, in a more general sense, combinatorial) optimiser.

Most of the insights obtained from visualisation seemed to give more and more credit to Poli and McPhee's concept of Homologous Crossover [226], where crossover preserves information already shared between candidates. The problem observed with the K-GPSO-GPU algorithm is that the global best weighted score is often so great in comparison, that it is simply duplicated due to the weighted crossover operator. It should be noted again that the Karva language was intended for multiple genetic operators, however. This may serve to reduce the excessive loss of diversity in the GPSO algorithm.

8.4 Domain-specific Languages for ABM

A new domain-specific language named MOL designed specifically for lattice-oriented agent-based models was presented in Chapter 5. The major aspects of this language which make it well suited to ABM include its use of multi-stage programming to provide high performance and extensibility without compromise. It is also able to accept extensions written in C/C++ thanks to Clang. Terra is an example of what a well-executed multi-stage programming language can achieve, and it is truly impressive.

Good results from simple MOL programs were achieved, which initially had inexplicably outperformed semantically identical C++ code, compiled with the GNU C++ compiler. It was not until the LLVM internal representation (IR) code was obtained from disassembling the Terra code that it became clear why this happened. LLVM's automatic optimisations include code vectorisation, which in this case had automatically generated SIMD AVX instructions, whereas the C++ code was not vectorised.

At this point in time, the MOL language is in infancy. The language is restricted to lattice-oriented models (spatial ones are very restrictive), and lack a number of syntactic improvements. A significant amount of work remains to expose the system to the public. In particular, it was found that debugging extensions written in Terra is exceptionally difficult. The low-level nature of Terra sometimes cause unhelpful run-time errors, and the perils of memory corruption as a result are also not infrequent. Debugging support would need to be well implemented for this system to be usable by the public.

Although some problems are still to be addressed, the language appears to be well suited to describing lattice-oriented agent-based models. To the best of the author's knowledge, no other compiled multi-stage agent-based modelling language exists at the time of writing.

Specific aspects of the MOL language were carefully chosen to facilitate the addition of an optimiser. Specifically, the choice of a multi-stage architecture was to enable run-time code generation, and therefore the recombination, reconstruction and run-time compiling of code for candidate solutions to specific objective functions.

8.5 Parallel Domain-specific Optimisation in ABM

Chapter 6 considers the addition of both high performing data-parallelism as well as a program-space optimiser. It was shown that it is indeed viable to optimise written code, and portions of it towards a specified objective function. Though, along the lines of the usual issues associated with using evolutionary algorithms, a number of issues arose during development.

Firstly, the problem of providing terminals and nonterminals (the search space) for an algorithm to effectively search through is difficult. In MOL, a user provides a section of code that is either recombined, or permuted (or a single statement is extracted). It is possible that a user may provide all but one statement necessary for an optimal solution, and therefore, they will receive a suboptimal solution. A similar problem exists in general with evolutionary algorithms such as Genetic Programming.

A separate problem is that of the objective function. In many evolutionary algorithms, providing an objective function is as much of an art as providing terminals and nonterminals, and even to the point where it may compete with the effort involved in hand-producing a suitable (but probably suboptimal) solution. In cases like these, one is essentially trading one complexity for another. It is worth noting however, that evolutionary algorithms still certainly have their place, and have resulted in surprising results particularly in co-evolution literature [170, 167].

Though the problem of functions, terminals and objective functions still produce undesirable effects in MOL programs with uncertainty, the problem of the search space is another issue which has been handled to some extent. Quite often in evolutionary algorithms, the search space is simply too large. The number of combinations of a set of functions and terminals can become very large, and coupled with problems such as search space neutrality, optimisers struggle tremendously to converge or bias towards good solutions. The advantage of allowing a user to specify precisely the area in which they require structural optimisation assists grealy in reducing the search space. The rationale for this is that frequently a user may know precisely which aspects of a model are certain, which is not necessary to include in the optimisation efforts. In custom-written source code, it is all too easy to subject an entire model to the same process. In MOL it is also quite easy to modify the search space tremendously by simply moving one line of code, should a previous attempt to optimise a section of code not be successful.

Another consideration is that of the timeframe required for an optimiser to obtain a reasonable solution. In the modelling process, any time lost to an optimiser is undesirable, so it therefore follows that it should be minimised as much as possible. Unfortunately, issues such as stochasticity and model complexity significantly increase computational cost. It is for this reason that the use of graphics processing hardware was pursued. Some problems such as lattice size configurations demanded that different parallelisation strategies be made, but fortunately, it is possible for the underlying code to pre-determine a suitable strategy given the size of the lattices.

But perhaps a greater issue was that of concurrency and its related resource contention characteristics. Graphics processing units were built to cater for large quantities of pixel data, not scientific data. As a result, the after-thought of commodity-priced parallel scientific computing placed several restrictions on code design, with regards to memory hierarchies and synchronicity. These issues were reduced significantly with the advent of Compute Unified Device Architecture (CUDA) and related technologies. The specific issues encountered in MOL were those of race conditions on lattice sites and model scores. In an effort to eliminate these, checkerboard (or "red-black") update patterns were implemented, along with more coarse patterns (3x3 grid blocks per thread) to cater for models in which read/write access were necessary for neighbour sites.

An additional issue which is not to be ignored is that of the parameter set necessary to operate the MOL optimiser. Being an evolutionary optimiser, there are mutation and crossover probabilities, the *k*-expression head length, and candidate model numbers (population size). In addition, it is also necessary to specify a lattice size for the model, suitable for the parallelisation strategy chosen. While at this point a user must choose a strategy manually, this can be automated in the future.

8.6 Photobioreactor Modelling

Chapter 7 presents a thorough study on the use of ABM for algal photobioreactor modelling. The purpose of introducing this modelling sample was to facilitate a thorough study of MOL and its utility in developing a sophisticated model, and assisting the user to obtain useful inspiration from code-based optimisation. The same model was first implemented in a conventional manner, using C++.

A considerable issue which is the likely cause of infrequent application of ABM in algaculture is that of computational complexity. Without some form of parallelism or super-agent strategy [252], it would be impossible to model more than 1mL of undried biomass. While possible that a whole bioreactor could be simulated using a powerful cluster computer, the purpose in considering graphics processing units for this task is the dramatically lower cost and high theoretical throughput. Such devices are now common in laboratory bench desktop computers, and while they may not provide enough power to simulate an entire reactor, they are sufficient to simulate smaller reactors which are still very much able to provide valuable insights.

The useful ability of MOL to compile for graphics processing hardware for both optimisation and automatic parallelism of sophisticated models such as these is very encouraging. Automatic parallelism has been under intense investigation for some time [221]. The most influential factor behind this movement is the fact that chip manufacturers are no longer increasing clock speeds, but rather focussing on the addition of extra cores on general purpose processors [161]. It is for this reason that parallelism is becoming more important in the context of high performance applications.

As a final remark, it is useful to consider this system in the context of emergence engineering. Though the term is somewhat vague, the author believes what it refers to in the general sense is a field that deserves more investigation. The invention of algorithms for explaining flocking and schooling behaviour in the 1970s and 80s were very enlightening, and concepts previously thought fundamentally unexplainable are being unmasked slowly. Interest in larger systems in recent years which were previously out of reach now show great promise. Such ideas are at the forefront of scientific inquiry, and while optimisation algorithms and technologies such as MOL and others may be relatively deterministic in the sense that they must be operated wisely, the very least they can accomplish is inspiration leading to great scientific discoveries in the years to come.

CHAPTER 9 CONCLUSIONS

To conclude this dissertation, the original objectives are reiterated and findings are presented. In Section 9.1 a brief overview of the thesis as a whole is presented. Section 9.2 presents findings regarding Domain-Specific Languages (DSLs) and their use in the context of ABM. Work towards high performance ABM is then concluded in Section 9.3, and finally, findings from the main objective of the dissertation are presented in Section 9.4. A summary is provided in Section 9.5. Opportunities for future work are numerous, and some of these are outlined in Section 9.6.

9.1 Overview

The main overarching objectives of this dissertation were to facilitate a single objective. It was to arrive at a DSL which is capable of optimising model structure itself using a preprocessor-style optimiser and special syntax to ease its use. The purpose of which was twofold. First, it was to ease modelling by reducing or eliminating tedious trial-and-error processes and secondly to automatically generate inspiration from hypotheses manifested by hand-written objective functions. Neither of these would be possible in this manner without also preventing the architecture from succumbing to heavy abstraction penalties. However, it is not a strict requirement to use the structural optimisation algorithms within the language. In fact, a useful auxiliary attribute of this language is that it is a high-performance platform for any lattice-oriented agent-based model.

As mentioned above, two considerable problems needed to be addressed before such an architecture could be constructed. Firstly, the implementation of a domain-specific language with the ability to describe models and selective optimisation problems with ease and clarity. The second problem was to mitigate the excessive computational complexity of model optimisation.

9.2 Domain-specific Languages for ABM

Domain-specific languages for ABM had already been discussed in the context of ABM, but none used the multi-stage programming paradigm of Taha [274]. An enormously helpful aspect of the multi-stage programming mindset is that of code specialisation, and the ease of implementing domain-specific languages. One such language named Terra, a very recent language, was investigated in detail. It was found that Terra is extremely well suited to constructing and maintaining a domain-specific language. Particular aspects of it which were very helpful include its use of LLVM (a mature compiler architecture), run-time code generation (RTCG), its design goals of low-level support and high performance, and seamless integration into a high-level dynamically typed LuaJIT language (high performance Lua) environment.

A new language named MOL was developed in Chapter 5, given with examples. Chapter 6 presented a considerable enhancement of the DSL to incorporate an optimiser. Examples were given, and a short performance comparison with a spatial version was made with conventional code, which showcased the considerable advantages of LLVM in high-level optimisations of its internal representation (IR, LLVM's single static assignment intermediate code). Another advantage is the ability of MOL to specify models using extensions written in Terra, or even C/C++ using Clang through Terra. This is a formidable advantage, considering that this allows easy extension not only using specialising extensions compiled to machine code, but also other third party libraries.

In summary, the MOL language shows great potential as a modelling platform on its own. Much work remains to ready the platform for public use, however. Cosmetic and technical features such as a well-designed user interface and debugging tools are among the most important. For the purpose of model structure optimisation, this language incorporates the most important features for both high performance and modifiable syntax trees.

9.3 High Performance ABM and Optimisation

Chapters 2, 3 and 4 for the most part, contributed directly to the implementation of optimisation techniques using data-parallel algorithms in the context of ABM. Auxiliary objectives of these chapters included effective visualisation (see Sections 3.6 and 4.8), and optimiser calibration (see Section 3.5). These techniques were found very useful for improving algorithms.

High performance ABM was investigated in Chapter 2. This resulted in the combination of well-known spatial partitioning techniques and graphics processing hardware. The result was a simulation which could simulate in real time around 1 million interactive agents for an agent-based model given the right conditions. This was important due to the increasing interest from the ABM community in large systems, but also because later optimisation efforts of this work were notorious for excessive computational complexity. It was found that interactive agent-based models which have sufficiently small communication radii respond well to spatial partitioning algorithms, especially when agents are uniformly distributed. However, further research is necessary to adapt or to find a hybrid algorithm to deal with situations where ideal conditions are partially, or not at all, satisfied. In particular, when agents cluster tightly, an algorithm such as the uniform grid breaks down in performance very quickly.

A number of conclusions can be made from Chapters 3 and 4 (Continuous and Combinatorial Optimisation). Specifically with regard to continuous global optimisation, evolutionary optimisers have shown themselves to be capable of solving standard test functions. The addition of more advanced space exploration techniques are of use, albeit limited, at least in the experiments conducted as part of this research. Apart from the usual Lévy flights, similar stable distributions were also considered, with unfortunately similar results. Their utility may be better founded in other problem domains, however.

The use of data-parallelism in these optimisers were particularly successful, and a particular focus was given to the *many-optimising-liaisons* PSO (MOLPSO) and the Firefly Algorithm. While the PSO in general has enjoyed research effort towards parallelisation, the FA has not. Results indicate that considerable performance increases are obtainable from parallel versions of these algorithms, due to their inherent parallel nature. The interactive nature of the FA did complicate efforts significantly, however. The MOLPSO was investigated for its simple and elegant algorithm, as well as its small memory footprint, which are desirable attributes in the context of data-parallelism. The FA was also considered, since it is commonly perceived as a more complex optimiser than the PSO due to its use of a reinterpreted Lambert-Beer law [155] for decaying observed fitness levels. Like many other metaheuristics, both of these algorithms have problem domains in which they are more suited.

Brief experimentation was carried out in the investigation of meta-optimisation in Chapter 2. It was found that while automatic parameter calibration of optimisers are useful, they appear to suffer tremendously from excessive computing times. Apart from using graphics hardware to accelerate this, no other attempts to improve performance were made other than to resign to using small systems. The characterisation of the process in terms of fitness-by-generation across the different test functions were so varied that it indicated the problem of overfitting was indeed present. The practical use of this experimentation was to arrive at suitable parameters for later use. The performance characteristics of this experiment also alluded to similar problems with regard to model optimisation. This was especially clear when the sub-optimiser was reinterpreted as a model with input parameters and output behaviours, given that it was inspired from an agent-based model.

Chapter 3 concluded with a brief discussion of visualisation techniques, useful for gaining a qualitative sense of the algorithm's operation. Higher dimension systems were particularly difficult to visualise, but a very simple technique was implemented which gives an adequate indication of both variation and specific desirable system states (in particular, global optima).

Chapter 4 considered the related problem of combinatorial optimisation in the context of programs and expression trees. It was found that a linear representation named *Karva* was very elegant and well suited to parallelisation. As a first experiment, the venerable Genetic Programming algorithm was adapted to use *Karva*-expressions and graphics processing hardware, and then tested against selected problems. While good performance was achieved, it was prudent to consider more recent developments in the field of metaheuristics for this purpose. The geometric unification framework of Moraglio [189] provided the necessary toolset to potentially further the synergy between the related field of continuous and combinatorial optimisation. These divided disciplines have received separate development efforts in the past. It followed that the cross-discipline synergy may follow. Both the PSO and Firefly Algorithm (FA) were considered in the light of the geometric framework informally using Karva, and adapted accordingly. Unfortunately the Karva-based PSO failed to impress in the context of expression tree evolution. It is thought this resulted from the lack of higher quality genetic operators on Karva expressions, as Karva was initially developed with several operators, instead of simple point-mutation and one-point crossover.

The geometric FA provided impressive results, however. Its use of fitness decay (reminiscent of the original FA) seemed to produce some synergy from the use of the Hamming distance metric. In fact, a particular value of the decay parameter (0.008) appeared to provide the best performance and highest chance of success on a symbolic regression problem. In summary, while the original Genetic Programming algorithm may still be of use in this context, it is less readily parallelisable, and great benefit is obtainable from the combination of Karva, graphics processing units, and interactive optimisers such as the FA using geometric unification.

Chapter 4 concluded with a discussion of two techniques to visualise program-space. Both are simple techniques but have their respective drawbacks. It is far more common to see a visualisation pertaining to the fitness evaluation of a set of candidates, but not as common to see a visualisation of the population's structural composition. Though computationally expensive, these techniques can assist in improving optimisers and gaining a sense of how they operate.

9.4 Model Structures, Parameters, and Calibration

Chapter 6 made the necessary enhancements and additions to the MOL language presented in Chapter 5 using considerations from earlier chapters to arrive at a ABM language with an embedded structural optimiser.

Data-parallelism was also integrated into the MOL language, which, while sharing the parser and type checker, resulted in a different code generator, as can be expected. The resulting system was able to generate Terra code, which could then be compiled using either the official NVIDIA CUDA LLVM backend through Terra, or a standard X86 LLVM backend for the processor architectures used during testing.

Three different optimisation configurations were developed: single for single-statement selection, permutation for multi-statement selection with replacement, and recombination for a complete deconstruction and recombination of code. The last configuration mentioned uses the Karva expression language discussed in Chapter 4, and related optimisers. For this reason, the last configuration is given the most investigation.

It was found that given an appropriate selection of parameters, (such as population size, mutation/crossover probabilities, *k*-expression head lengths), suitable program structures can certainly be generated for a given objective function to be minimised, or maximised. Performance seemed to be adequate, except in the case where population sizes exceed 256. In these cases, it appears that CUDA compilation takes an excessive amount of time. The faster construction of MOL programs on CPU meant that smaller population sizes were sometimes better computed on the CPU. The most convincing results collected from MOL was the successful recombination of a simple set of terminals and nonterminals to solve the Santa Fe Ant Trail problem. The resulting trees were not without redundancies, but considering the complexity of the problem, such a result is impressive.

Chapter 7 presented a comparison of the MOL language and a more conventional C/C++ custom coding process for algal photobioreactors. Considerable effort was made to arrive at an agent-based model with a realistic set of components, reminiscent of what an actual modelling process would generate. The model developed of a flat-plane photobioreactor was then reimplemented using MOL, and compared with the conventional approach. It was found that model-specific extensions needed to be written to implement the more complex features of the model. It is anticipated that such extensions would be stored in problem-domain specific libraries. This will allow experts to propagate their expertise in the form of specialising code.

A sample optimisation problem was investigated in the photobioreactor model. Performance was found to be adequate to maintain a population of 64 candidate photobioreactor models. The seemingly opposing objectives of increasing illumination on cells and the increase of biomass was the objective function. The purpose of this was to see if the algorithm is capable of generating a code sample which is able to satisfy such an objective. While clearly a difficult problem, results were quite interesting. Semantic meanings of the initial state transition code were somewhat lost, however, they seemed to take on new meanings. Symbol histograms of populations across several generations and a fitness plot by generation confirmed that the algorithm was indeed converging, as more and more candidates were sharing the same symbols in the same *k*-expression indices.

9.5 Summary

In summary, the language developed is another complimentary tool to the considerable set of ABM modelling apparatus. Automatic parallelism and structural optimisation allow the user to explore different simulation configurations, obtain solutions for tedious problems, and simulate larger systems. The architecture on which MOL is built is an incredibly malleable and extensible platform, which will surely attract efforts to incorporate large third party libraries. *In conclusion, the research shows that the use of data-parallelism by multi-stage programming in a simple domain-specific language for high performance and extensibility are sufficient to give rise to the compute intensive but beneficial task of automatic model structure optimisation.*

9.6 Future Work

There is scope for expanding on this research in multiple areas. Firstly, it is still relevant to consider spatial partitioning in the context of MOL. This would change the architecture significantly, and given the correct specialising code, it could be included automatically when needed. Secondly, the optimisers currently employed in the language could benefit from more sophisticated genetic operators, or perhaps a completely different algorithm such as Grammatical Evolution [249].

There is indeed an opportunity to extend the language by adding more optimisation constructs other than single, permutation and recombination. Along the lines of Grammatical Evolution, it may be worth considering a "deep" reconstruction of code where even conditions are decomposed into terminals, along with random variables.

In addition to other configurations, it would be beneficial to hybridise continuous and combinatorial optimisation in order to construct expressions such as between 0.1 and 0.3 for constrained continuous optimisation alongside structural optimisation. An extension of the Gene Expression Programming algorithm by Ferreira [64] is capable of evolving constants in this manner.

A significant shortcoming of MOL is that a potential solution must be provided in order to facilitate a search. In cases when the user cannot provide one, or can only provide a subset of the terminals and nonterminals needed, the algorithm can only produce suboptimal results. The author admits that this problem is related to that of the selection of terminals and nonterminals in the usual usage of Genetic Programming, and future research in this area would be extremely helpful.

Further extensions of the MOL language could be to exploit the use of Terra's support for multiple DSLs in one script file. It would be useful to consider the addition of a syntax specifically for finite state machines, or perhaps to allow X-Machine style agent descriptions.

Pre-compiling library code can also be useful. Significant portions of the Terra libraries themselves are pre-compiled using LuaJIT. The advantage of this is a faster loading time, which given the size of the MOL libraries, could significantly improve startup time.

It may be useful in future, to abstract the concept of structural optimisation to the alteration of Terra syntax trees directly. At present, the MOL language is compiled to Terra syntax trees, and then the optimiser alters these trees. It may be worthwhile to consider altering Terra syntax trees directly. Of course, generality in this sense pays a considerable price. Having a specialised DSL to limit the search space is very important for this research. However, it may still be useful when applying the same concepts to other DSLs, not necessarily related to ABM.

BIBLIOGRAPHY

[1] B. G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient simulation of agent-based models on multi-GPU and multi-core clusters. In *Proc. SIMUTools 2010, 3rd International ICST Conference on Simulation Tools and Techniques, Torremolinos, Malaga, Spain*, pages 1–10, 15-19 March 2010.

[2] Apostolopoulos and Vlachos. Application of the firefly algorithm for solving the economic emissions load dispatch problem. *International Journal of Combinatorics*, 1, 2011.

[3] A. W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6:85–103, 1985.

[4] D. Augusto and H. J. C. Barbosa. Symbolic regression via genetic programming. In *Neural Networks, 2000. Proceedings. Sixth Brazilian Symposium on*, pages 173–178, 2000.

[5] P. Aungkulanon, N. Chai-ead, and P. Luangpaiboon. Simulated manufacturing process improvement via particle swarm optimisation and firefly algorithms. *Proc. Int. Multiconference of Engineers and Computer Scientists*, 2:1123–1128, 2011.

[6] R. Axelrod. The emergence of cooperation among egoists. *The American Political Science Review*, 75:306–318, 1981.

[7] R. Axelrod. Advancing the art of simulation in the social sciences. *Complexity*, 3(2):16–22, 1997.

[8] R. Axelrod. The dissemination of culture: a model with local convergence and global polarization. *J. Conflict Resolution*, 41:203–226, 1997.

[9] S. K. Azad and S. K. Azad. Optimum design of structures using an improved firefly algorithm. *International Journal of Optimization in Civil Engineering*, 1:327–340, 2011.

[10] J. Bachrach, J. McLurkin, and A. Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1175–1178. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[11] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[12] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[13] C. Bastos-Filho, M. O. Junior, and D. Nascimento. Running particle swarm optimization on graphic processing units. In N. Mansour, editor, *Search Algorithms and Applications*. InTech, 2011. ISBN: 978-953-307-156-5.

[14] B. Basu and G. K. Mahanti. Firefly and artificial bees colony algorithm for synthesis of scanned and broadside linear array antenna. *Progress in Electromagnetic Research*, 32:169–190, 2011.

[15] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[16] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 436–501. IGI Global, August 2013.

[17] J. Beal and R. Schantz. A spatial computing approach to distributed algorithms. In *45th Asilomar Conference on Signals, Systems, and Computers*, 2010.

[18] J. Beal, K. Usbeck, and B. Krisler. Lightweight simulation scripting with proto. *Spatial Computing 2012 colocated with AAMAS*, page 1, 2012.

[19] E. W. Becker. *Microalgae: biotechnology and microbiology*. Cambridge University Press, 1994.

[20] A. Beer. Bestimmung der absorption des rothen lichts in farbigen flüssigkeiten. *Annalen der Physik und Chemie*, 86:78–90, 1852.

[21] H.-G. Beyer, H.-P. Schwefel, and I. Wegener. How to analyse evolutionary algorithms. *Theoretical Computer Science*, 287(1):101 – 130, 2002. Natural Computing.

[22] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(13):217 – 239, 2005.

[23] D. Birks, M. Townsley, and A. Stewart. Generative explanations of crime: Using simulation to test criminological theory. *Criminology*, 50:221–254, 2012.

[24] E. Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3):7280–7287, 2002.

[25] M. Brameier. *On Linear Genetic Programming*. PhD thesis, University of Dortmund, 2004.

[26] M. Brameier and W. Banzhaf. *Explicit control of diversity and effective variation distance in linear genetic programming*. Springer, 2002.

[27] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer Verlag, 1983.

[28] B. Calvez and G. Hutzler. Automatic tuning of agent-based models using genetic algorithms. In *Proceedings of the 6th International Workshop on Multi-Agent Based Simulation (MABS 2005)*, pages 41–57, 2005.

[29] B. Calvez, G. Hutzler, et al. Adaptative dichotomic optimization: a new method for the calibration of agent-based models. In *A. Tanguy C. Bertelle J. Sklenar et G. Fortino, éditeurs, Proceedings of the 2007 European Simulation and Modelling Conference (ESM07)*, pages 415–419, 2007.

[30] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel Computing*, 21(5):803–823, 1995.

[31] A. Cano, J. L. Olmo, and S. Ventura. Parallel multi-objective ant programming for classification using GPUs. *J.Parallel and Distributed Computing*, 73:713–728, 2013.

[32] T. Castle and C. G. Johnson. Evolving high-level imperative program trees with strongly formed genetic programming. In *Proceedings of the 15th European Conference on Genetic Programming, EuroGP*, volume 7244, pages 1–12. Springer, April 2012.

[33] J. M. Chambers, C. L. Mallows, and B. W. Stuck. A method for simulating stable random variables. *Journal of the American Statistical Association*, 71:340–344, 1976.

[34] A. Chatterjee, G. K. Mahanti, and A. Chatterjee. Design of a fully digital controlled reconfigurable switched beam coconcentric ring array antenna using firefly and particle swarm optimization algorithms. *Progress in Electromagnetic Research*, B:113–131, 2012.

[35] Y. Chisti. *Airlift Bioreactors*. Elsevier Applied Science, 1989.

[36] D. M. Chitty. Fast parallel genetic programming: multi-core cpu versus many-core GPU. *Soft. Comput.*, 16:1795–1814, 2012.

[37] S. Coakley, R. Smallwood, and M. Holcombe. Using X-machines as a formal basis for describing agents in agent-based modelling. In *Proceedings of the 2006 Spring Simulation Multiconference*, 2006.

[38] Collier. Repast: An extensible framework for agent simulation. Technical report, Social Science Research Computing, University of Chicago, 2003.

[39] D. N. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, Department of Electrical Engineering and Computer Science, 1999.

[40] M. Crosbie and E. H. Spafford. Applying genetic programming to intrusion detection. Technical report, Department of Computer Sciences, Purdue University, West Lafayette, 1995. AAAI Technical Report FS-95-01.

[41] L. Cupertino, C. Silva, D. Dias, M. A. Pacheco, and C. Bentes. Evolving CUDA PTX programs by quantum inspired

linear genetic programming. In *Proceedings of GECCO'11*, 2011.

[42] K. Czarnecki, J. T. ODonnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, template Haskell, and C++. In *Domain-Specific Program Generation*, pages 51–72. Springer, 2004.

[43] J. M. Daida, A. M. Hilss, D. J. Ward, and S. L. Long. Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6, 2005.

[44] P. Danilewski, S. Popov, and P. Slusallek. Binned SAh Kd-tree construction on a GPU. Technical report, Saarland University, Germany, 2010. 15 Pages.

[45] S. Das. On agent-based modeling of complex systems: Learning and bounded rationality. Tech Report, 2006.

[46] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, pages 105–116, 2013.

[47] M. d'Iverno and M. Luck. *Understanding Agent Systems*. Springer, 2001. ISBN 3-540-41975-6.

[48] M. Dorigo and T. Stützle. Ant colony optimization: Overview and recent advances. Technical Report ISSN 1781-3794, TR/IRIDIA/2009-013, Free University of Brussels, Av F. D. Roosevelt 50, CP 194/6 1050 Bruxelles, Belgium, May 2009.

[49] J. Drieseberg and C. Siemers. C to cellular automata and execution on CPU, GPU and FPGA. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 216–222. IEEE, 2012.

[50] A. Drogoul, D. Vanbergue, and T. Meurisse. Multi-agent based simulation: Where are the agents? In *Multi-agent-based simulation II*, pages 1–15. Springer, 2003.

[51] R. M. D'Souza, M. Lysenko, S. Marino, and D. Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 21. Society for Computer Simulation International, 2009.

[52] K. Durkota. Implementation of a discrete firefly algorithm for the QAP problem within the Seage framework. Technical report, Czech Technical University, 2011.

[53] R. Dutta, R. Ganguli, and V. Mani. Exploring isospectral spring-mass systems with firefly algorithm. *Proc. Roy. Soc. A.*, 467:‘, 2011.

[54] A. M. Edwards. Overturning conclusions of Lévy flight movement patterns by fishing boats and foraging animals. *Ecology*, 92(6):1247–1257, 2011.

[55] A. Eiben, P.-E. Raué, and Z. Ruttkay. Genetic algorithms with multi-parent recombination. In *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, 1994.

[56] A. E. Eiben. Multi-parent recombination. *Evolutionary computation*, 1:289–307, 1997.

[57] P. Eilers and J. Peeters. A model for the relationship between light intensity and the rate of photosynthesis in phytoplankton. *Ecological Modelling*, 42:199–215, 1988.

[58] P. Eilers and J. Peeters. Dynamic behaviour of a model for photosynthesis and photoinhibition. *Ecological Modelling*, 69:113–133, 1993.

[59] J. M. Epstein. Agent-based computational models and generative social science. *Generative Social Science: Studies in Agent-Based Computational Modeling*, pages 4–46, 1999.

[60] J. M. Epstein, R. Axtell, and P. 2050. *Growing Artificial Societies : Social Science From the Bottom up.* Complex Adaptive Systems. Brookings Institution Press, 1996.

[61] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[62] J. Ferber. *Multi-Agent Systems - An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[63] C. Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[64] C. Ferreira. *Gene Expression Programming*, volume 21 of *Studies in Computational Intelligence*. Springer-Verlag, Berlin Heidelberg, 2 edition, 2006. ISBN 3-540-32796-7.

[65] J. Ferrer, C. Prats, and D. López. Individual-based modelling: An essential tool for microbiology. *J Biol Phys*, 34:19–37, 2008.

[66] G. P. Figueredo, P.-O. Siebers, and U. Aickelin. Investigating mathematical models of immuno-interactions with

early-stage cancer under an agent-based modelling perspective. *BMC Bioinformatics*, 14(6):1–38, 2013.

[67] G. P. Figueredo, P.-O. Siebers, U. Aickelin, and S. Foan. A beginner's guide to systems simulation in immunology. In C. A. Coello Coello, J. Greensmith, N. Krasnogor, P. Liò, G. Nicosia, and M. Pavone, editors, *Artificial Immune Systems*, volume 7597 of *Lecture Notes in Computer Science*, pages 57–71. Springer Berlin Heidelberg, 2012.

[68] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. Technical report, Intel Software Solutions Group, Intel Corporation, May 2008.

[69] I. K. Fodor. A survey of dimension reduction. Technical Report UCRL-ID-148494, Lawrence Livermore National Laboratory, 2002.

[70] E. Franchi. A domain specific language approach for agent-based social network modeling. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 607–612, 2012.

[71] M. Galassi, B. Gough, and G. Jungman. *GNU Scientific Library Reference Manual*. Network Theory Limited, 2009.

[72] A. H. Gandomi and X.-S. Yang. Mixed variable structural optimization using firefly algorithm. *Computers and Structures*, 89:2325–2336, 2011.

[73] V. Garcia, M. Birbaumer, and F. Schweitzer. Testing an agent-based model of bacterial cell motility: How nutrient concentration affects speed distribution. *The European Physical Journal B*, 82:235–244, 2011.

[74] M. Garcia-Malea, C. Brindley, E. D. Rìo, F. Acién, J. Fernández, and E. Molina. Modelling of growth and accumulation of carotenoids in haematococcus pluvialis as a function of irradiance and nutrients supply. *Biochemical Engineering Journal*, 26:107–114, 2005.

[75] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, October 1970.

[76] C. Gerber, J. Siekmann, and G. Vierke. Holonic multi-agent systems. Technical Report RR-99-03, Deustches Forschungzentrum für Künstliche Intelligenz GmbH, May 1999.

[77] A. P. Gerdelan. *Fuzzy Motion Controllers and Hybrids*. PhD thesis, Computer Science, Massey University, Albany, New Zealand, 2011.

[78] L. Gherardi, D. Brugali, and D. Comotti. A Java vs. C++ performance evaluation: a 3D modeling benchmark. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. Springer, 2012.

[79] G. Giannakouris and V. V. G. Dounias. Experimental study on a hybrid nature-inspired algorithm for financial portfolio optimization. *SETN 2010, LNAI 6040*, 1:101–111, 2010.

[80] N. Gilbert and P. Terna. How to build and use agent-based models in social science. *Mind & Society*, 1:57–72, 2000.

[81] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Company Inc, 1989.

[82] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[83] T. E. Gorochowski, A. Matyjaszkiewicz, T. Todd, N. Oak, K. Kowalska, S. Reid, K. T. Tsaneva-Atanasova, N. J. Savery, C. S. Grierson, and M. di Bernardo. BSim: An agent-based tool for modeling bacterial populations in systems and synthetic biology. *PLoS ONE*, 7(8), August 2012.

[84] S. Green. Particle simulation using CUDA. Technical report, NVIDIA, May 2010.

[85] E. Greenwald. A stochastic model of algal photobioreactors. Master's thesis, Ben-Gurion University of the Negev, 2010.

[86] C. Grelck, F. Penczek, and K. Trojahner. CAOS: A domain-specific language for the parallel simulation of cellular automata. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 4671 of *Lecture Notes in Computer Science*, pages 410–417. Springer Berlin Heidelberg, 2007.

[87] V. Grimm and S. F. Railsback. *Individual-based Modeling and Ecology*. Princeton University Press, 2005.

[88] V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. Thulke, J. Weiner, T. Wiegand, and D. L. DeAngelis. Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. *Science*, 310:987–991, 2005.

[89] L. J. Gross, K. A. Rose, E. Rykiel, W. van Winkle, and E. E. Werner. Individual-based modeling: Summary of a workshop. In DeAngelis and Gross, editors, *Individual-based Models and Approaches in Ecology*. Chapman and Hall, 1992.

[90] M. Guerin, M. E. Huntley, M. Olaizola, et al. Haematococcus astaxanthin: applications for human health and nutrition. *TRENDS in Biotechnology*, 21(5):210–216, 2003.

[91] C. Hahn. A domain specific modeling language for multiagent systems. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 233–240. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[92] K. Hawick. Visualising multi-phase lattice gas fluid layering simulations. In *Proc. International Conference on Modeling, Simulation and Visualization Methods (MSV'11)*, pages 3–9, Las Vegas, USA, 18-21 July 2011. CSREA.

[93] K. A. Hawick. 3D visualisation of simulation model voxel hyperbricks and the cubes program. Technical Report CSTN-082, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand, October 2010.

[94] K. A. Hawick. Static and dynamical equilibrium properties to categorise generalised game-of-life related cellular automata. In *Int. Conf. on Foundations of Computer Science (FCS'12)*, pages 51–57, Las Vegas, USA, 16-19 July 2012. CSREA.

[95] K. A. Hawick and A. V. Husselmann. Photo-penetration depth growth dependence in an agent-based photobioreactor model. Technical Report CSTN-204, Computer Science, Massey University, Auckland, New Zealand, Las Vegas, USA, July 2013.

[96] K. A. Hawick, H. A. James, and C. J. Scogings. Grid-boxing for spatial simulation performance optimisation. In T. Znati, editor, *Proc. 39th Annual Simulation Symposium*, pages 98–106, Huntsville, Alabama, USA, 2-6 April 2006. The Society for Modeling and Simulation International, IEEE Computer Society.

[97] K. A. Hawick, A. Leist, D. P. Playne, and M. J. Johnson. Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators. In *9th Australasian Symposium on Parallel and Distributed Computing(AusPDC)*, volume 118, pages 3–12, Perth, Australia, 17-20 January 2011. CRPIT.

[98] K. A. Hawick and D. P. Playne. Halo gathering scalability for large scale multi-dimensional Sznajd opinion models using data parallelism with GPUs. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, pages 95–101, Las Vegas, USA, 16-19 July 2012. CSREA.

[99] K. A. Hawick and D. P. Playne. Simulation software generation using a domain-specific language for partial differential field equations. In *11th International Conference on Software Engineering Research and Practice (SERP'13)*, number CSTN-187, page SER3829, Las Vegas, USA, 22-25 July 2013. WorldComp.

[100] K. A. Hawick, D. P. Playne, and C. J. Scogings. Simulating the generalised spatial Lotka-Volterra equations with multiple species on GPUs with automatic code generation. In *Proc. 12th IASTED Int. Conf. on Parallel and Distributed Computing and Networks (PDCN'13)*, pages 560–567, Innsbruck, Austria, 11-13 February 2013. IASTED.

[101] K. A. Hawick and C. J. Scogings. Hierarchical relationships and spatial emergence amongst multi-species animats. In *Proc. IASTED International Symposium on Modelling and Simulation (MS 2009)*, number 670-036, pages 1–6, Banff, Alberta, Canada, 6-8 July 2009. IASTED. CSTN-045.

[102] K. A. Hawick and C. J. Scogings. A minimal spatial cellular automata for hierarchical predator-prey simulation of food chains. In *Proc. International Conference on Scientific Computing (CSC'10)*, pages 75–80, Las Vegas, USA, 12-15 July 2010. WorldComp.

[103] K. A. Hawick and C. J. Scogings. Cycles, transients, and complexity in the game of death spatial automaton. In *Proc. International Conference on Scientific Computing (CSC'11)*, number CSC4040, pages 241–247, Las Vegas, USA, 18-21 July 2011. CSREA.

[104] K. A. Hawick, C. J. Scogings, and H. A. James. Defensive spiral emergence in a predator-prey model. *Complexity International*, 12(msid37):1–10, October 2008. ISSN 1320-0682.

[105] B. Heath, R. Hill, and F. Ciarallo. A survey of agent-based modeling practices (january 1998 to july 2008). *Journal of Artificial Societies and Social Simulation*, 12(4), October 2009.

[106] Helbing and Balietti. How to do agent-based simulations in the future: From modeling social mechanisms to emergent phenomena and interactive systems design. Technical report, Santa Fe Institute, 2011.

[107] J. H. Holland. *Adaptation in natural and artificial systems.* Ann Arbor: University of Michigan Press, 1975.

[108] J. H. Holland. Complex adaptive systems. *Daedalus*, 121(1):17–30, 1992.

[109] M. H. Horng. Vector quantization using the firefly algorithm for image compression. *Expert Systems with Applications*, 38, 2011.

[110] M. H. Horng and T. W. Jiang. The codebook design of image vector quantization based on the firefly algorithm. *Computational Collective Intelligence, Technologies and Applications*, 6426:438–447, 2010.

[111] G. Huang, Y. Long, and J. Li. Levy flight search patterns in particle swarm optimization. In *Seventh International Conference on Natural Computation*, 2011.

[112] Y. Hung and W. Wang. Accelerating parallel particle swarm optimization via GPU. *Optimization Methods and Software*, 27(1):33–51, 2012.

[113] A. V. Husselmann and K. Hawick. Intelligent individual agent-based simulation of photobioreactors and growth control. In *Proceedings of the 14th IASTED International Symposium on Intelligent Systems and Control (ISC 2013)*, 2013.

[114] A. V. Husselmann, K. Hawick, and C. Scogings. Model structure optimisation in lattice-oriented agent-based models. Technical Report CSTN-222, Computer Science, Massey University, 2014. Submitted to the International Journal of Modelling and Simulation (ACTA Press).

[115] A. V. Husselmann and K. A. Hawick. Simulating species interactions and complex emergence in multiple flocks of Boids with GPUs. In T. Gonzalez, editor, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 100–107, Dallas, USA, 14-16 Dec 2011. IASTED.

[116] A. V. Husselmann and K. A. Hawick. Spatial agent-based modelling and simulations - a review. Technical Report CSTN-153, Computer Science, Massey University, Albany, North Shore,102-904, Auckland, New Zealand, October 2011. In Proc. IIMS Postgraduate Student Conference, October 2011.

[117] A. V. Husselmann and K. A. Hawick. 3D vector-field data processing and visualisation on graphical processing units. In *Proc. Int. Conf. Signal and Image Processing (SIP 2012)*, number CSTN-140, pages 92–98, Honolulu, USA, 20-22 August 2012. IASTED.

[118] A. V. Husselmann and K. A. Hawick. Parallel parametric optimisation with firefly algorithms on graphical processing units. In *Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM'12)*, number 141 in CSTN, pages 77–83, Las Vegas, USA, 16-19 July 2012. CSREA.

[119] A. V. Husselmann and K. A. Hawick. Spatial data structures, sorting and GPU parallelism for situated-agent simulation and visualisation. In *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods (MSV'12)*, pages 14–20, Las Vegas, USA, 16-19 July 2012. CSREA.

[120] A. V. Husselmann and K. A. Hawick. Genetic programming using the Karva gene expression language on graphical processing units. Technical Report CSTN-171, Computer Science, Massey University, Auckland, New Zealand, July 2013.

[121] A. V. Husselmann and K. A. Hawick. Geometric optimisation using karva for graphical processing units. Technical Report CSTN-191, Computer Science, Massey University, Auckland, New Zealand, February 2013.

[122] A. V. Husselmann and K. A. Hawick. Levy flights for particle swarm optimisation algorithms on graphical processing units. *Parallel and Cloud Computing*, 2(2):32–40, April 2013.

[123] A. V. Husselmann and K. A. Hawick. Particle swarm-based meta-optimising on graphical processing units. In *Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC 2013)*, Phuket, Thailand, 10-12 April 2013. IASTED.

[124] A. V. Husselmann and K. A. Hawick. Random flights for particle swarm optimisers. In *Proc. 12th IASTED Int. Conf. on Artificial Intelligence and Applications*, Innsbruck, Austria, 11-13 February 2013. IASTED.

[125] A. V. Husselmann and K. A. Hawick. Simulating growth kinetics in a data-parallel 3D lattice photobioreactor. *Modelling and Simulation in Engineering*, 2013.

[126] A. V. Husselmann and K. A. Hawick. Towards high performance multi-stage programming for generative agent-based modelling. In *INMS Postgraduate Conference, Massey University*, October 2013.

[127] A. V. Husselmann and K. A. Hawick. Visualisation of combinatorial program space and related metrics. Technical Report CSTN-190, Computer Science, Massey University, Auckland, New Zealand, 2013.

[128] A. V. Husselmann and K. A. Hawick. Geometric firefly algorithms on graphical processing units. In *Cuckoo Search and*

Firefly Algorithm, pages 245–269. Springer, 2014.

[129] A. V. Husselmann and K. A. Hawick. Multi-stage high performance, self-optimising domain-specific language for spatial agent-based models. In *The 13th IASTED International Conference on Artificial Intelligence and Applications*, Innsbruck, Austria, February 2014. IASTED.

[130] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26. ACM, 2007.

[131] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua: An extensible extension language. *Software: Practice and Experience*, 26:635–652, 1996.

[132] G. K. Jati and S. Suyanto. Evolutionary discrete firefly algorithm for travelling salesman problem. *ICAIS2011, Lecture Notes in Artificial Intelligence (LNAI 6943)*, pages 393–403, 2011.

[133] K. Jim and C. Giles. Talking helps: evolving communicating agents for the predator-prey pursuit problem. *Artificial Life*, 6(3):237–254, 2000. Summer.

[134] R. Junges and F. Klügl. Evaluation of techniques for a learning-driven modeling methodology in multiagent simulation. In J. Dix and C. Witteveen, editors, *Multiagent System Technologies*, volume 6251 of *Lecture Notes in Computer Science*, pages 185–196. Springer Berlin Heidelberg, 2010.

[135] R. Junges and F. Klügl. Evolution for modeling - a genetic programming framework for SeSAm. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings*, 2011.

[136] R. Junges and F. Klügl. Programming agent behavior by learning in simulation models. *Applied Artificial Intelligence*, 26:349–375, 2012.

[137] R. Junges and F. Klügl. Learning tools for agent-based modeling and simulation. *Künstliche Intelligenz*, 27:273–280, 2013.

[138] M. Jureczko, M. Pawlak, and A. Mężyk. Optimisation of wind turbine blades. *Journal of Materials Processing Technology*, 167:463–471, 2005.

[139] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Meta-evolution in Graph GP. *Lecture Notes in Computer Science*, 1598:15–28, 1999.

[140] D. J. Kaup, T. L. Clarke, R. Oleson, and L. C. Malone. Crowd dynamics simulation research. In *Proc. 16th Conf. on Behaviour Representation in Modeling & Simulation (BRIMS'07), Orlando, FL, USA*, pages 173–180, 2007.

[141] K. Kawasaki. Diffusion constants near the critical point for time dependent Ising model I. *Phys. Rev.*, 145(1):224–230, 1966.

[142] Kennedy and Eberhart. Particle swarm optimization. *Proc. IEEE Int. Conf. on Neural Networks*, 4:1942–1948, 1995.

[143] A. Khadwilard, S. Chansombat, T. Thepphakorn, P. Thapatsuwan, W. Chainate, and P. Pongcharoen. Application of firefly algorithm and its parameter setting for job shop scheduling. *First Symposium on Hands-On Research and Development*, 1, 2011.

[144] J. T. O. Kirk. *Light and Photosynthesis in Aquatic Ecosystems*. Cambridge University Press, 2011.

[145] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[146] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[147] F. Klügl, R. Herrler, and M. Fehler. SeSAm: implementation of agent-based simulation using visual programming. In *AAMAS*, volume 6, pages 1439–1440, 2006.

[148] D. Kornhauser, U. Wilensky, and W. Rand. Design guidelines for agent based model visualization. *Journal of Artificial Societies and Social Simulation*, 12(2 1):1–32, 2009.

[149] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. Massachusetts Institute of Technology, 1992.

[150] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, June 1994.

[151] J. R. Koza, F. H. Bennett III, and O. Stiffelman. *Genetic Programming as a Darwinian Invention Machine*. Springer, 1999.

[152] A. V. Kravtsov, A. A. Klypin, and A. M. Khokhlov. Adaptive refinement tree: A new high-resolution N-Body code for cosmological simulations. *The Astrophysical Journal - Supplement Series*, 111:73–94, 1997.

[153] J.-U. Kreft, G. Booth, and J. W. Wimpenny. BacSim, a simulator for individual-based modelling of bacteria colony growth. *Microbiology*, 144:3275–3287, 1998.

[154] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[155] J. H. Lambert and E. Anding. *Johann Heinrich Lambert's Photometrie: photometria, sive de Mensura et Gradibus Luminis, Colorum et Umbrae*. W. Engelmann, 1892.

[156] W. B. Langdon. A many-threaded CUDA interpreter for genetic programming. In A. I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, and A. S. Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP*, pages 146–158. Springer, April 2010.

[157] W. B. Langdon. Graphics processing units and genetic programming: an overview. *Soft. Comput.*, 15:1657–1669, March 2011.

[158] L. Lardon, A. Hélias, B. Sialve, J.-P. Steyer, and O. Bernard. Life-cycle assessment of biodiesel production from microalgae. *Environmental science & technology*, 43(17):6475–6481, 2009.

[159] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[160] Y.-K. Lee and J. S. Pirt. Energetics of photosynthetic algal growth: influence of intermittent illumination in short (40 s) cycles. *Journal of General Microbiology*, 124(1):43–52, 1981.

[161] A. Leist. *Experiences in Data-Parallel Simulation and Analysis of Complex Systems with Irregular Graph Structures*. PhD thesis, Massey University, Auckland, New Zealand, November 2011.

[162] A. Leist, D. P. Playne, and K. A. Hawick. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience*, 21(18):2400–2437, 25 December 2009. CSTN-065.

[163] A. Lerner, Y. Chrysanthou, A. Shamir, and D. Cohen-Or. Data driven evaluation of crowds. In *Proceedings of the 2nd International Workshop on Motion in Games*, MIG '09, pages 75–83, Berlin, Heidelberg, 2009. Springer-Verlag.

[164] J. J. Liang and P. N. Suganthan. Dynamic multi-swarm particle swarm optimizer with local search. In *Evolutionary Computation*, 2005.

[165] LLVM. User guide for NVPTX back-end. http://llvm.org/docs/NVPTXUsage.html accessed 12 March, 2014.

[166] A. J. Lotka. *Elements of Physical Biology*. Williams & Williams, Baltimore, 1925.

[167] S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In J. R. Koza, W. Banzhaf, K. Chellapilla, D. Kumar, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the 3rd annual conference*, pages 214–222. Morgan Kaufmann, San Mateo, California, 1998.

[168] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[169] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A multiagent simulation environment. *Simulation*, 81:517–527, 2005.

[170] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler. Co-evolving soccer softbot team coordination with genetic programming. *RoboCup-97: Robot soccer world cup I*, 1:398–411, 1998.

[171] S. Luke and L. Spector. Evolving teamwork and coordination with genetic programming. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 150–156. MIT Press, 1996.

[172] M. Lysenko, R. M. D'Souza, and K. Rahmani. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[173] S. L. Lytinen and S. F. Railsback. The evolution of agent-based simulation platforms: A review of NetLogo 5.0 and ReLogo. In *Proceedings of the fourth international symposium on agent-based modeling and simulation (21st European Meeting on Cybernetics and Systems Research [EMCSR 2012])*, 2012.

[174] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation part 2: How to model with agents. In *Proc.*

2006 Winter Simulation Conference, Monterey, CA, USA, pages 73–83, 3-6 December 2006. ISBN 1-4244-0501-7/06.

[175] M. W. Macy and R. Willer. From factors to actors: Computational sociology and agent-based modeling. *Annual Review of Sociology*, 28, 2002.

[176] Madeira, E. Clua, A. Montenegro, and T. Lewiner. GPU Octrees and optimized search. *VIII Brazillian Symposium on Games and Digital Entertainment*, 2009.

[177] N. Mancuso. Firefly algorithm implementation. http://code.google.com/p/csc6810project/, 2010.

[178] B. B. Mandelbrot. The fractal geometry of nature. 1982. *San Francisco, CA*, 1982.

[179] S. M. Manson. Agent-based modeling and genetic programming for modeling land change in the Southern Yucatán peninsular region of Mexico. *Agriculture Ecosystems & Environment*, 111:47–62, 2005.

[180] M. Meissner, M. Schmuker, and G. Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural network training. *BMC Bioinformatics*, 7, 2006.

[181] J. Merchuk and X. Wu. Modeling of photobioreactors: Application to bubble column simulation. *Journal of Applied Phycology*, 15:163–170, 2003.

[182] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, 2005.

[183] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.

[184] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[185] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge, 2009. ISBN 9780521738330.

[186] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Santa Fe Institute, 1399 Hyde Park Rd. Santa Fe, NM 87501, June 1996.

[187] F. Mola, R. Miele, and C. Conversano. *Evolutionary Algorithms in Decision Tree Induction*, chapter 22, pages 443–468. I-Tech Education and Publishing, Vienna, Austria, 2008.

[188] M. Molga and C. Smutnicki. Test functions for optimization needs. Technical report, Univ. Wroclaw, Poland, 2005.

[189] A. Moraglio. *Towards a Geometric Unification of Evolutionary Algorithms*. PhD thesis, Computer Science and Electronic Engineering, University of Essex, 2007.

[190] A. Moraglio, C. D. Chio, and R. Poli. Geometric particle swarm optimization. In M. E. et al, editor, *Proceedings of the European conference on genetic programming (EuroGP)*, volume 4445 of *Lecture notes in computer science*, pages 125–136, Berlin, 2007. Springer.

[191] A. Moraglio and S. Silva. Geometric differential evolution on the space of genetic programs. *Genetic Programming*, 6021:171–183, 2010.

[192] A. Moraglio and J. Togelius. Geometric differential evolution. In *Proceedings of GECCO-2009*, pages 1705–1712. ACM Press, 2009.

[193] P. Moscato, C. Cotta, and A. Mendes. Memetic algorithms. In *New optimization techniques in engineering*, pages 53–85. Springer, 2004.

[194] W. Mulbry, S. Kondrad, and J. Buyer. Treatment of dairy and swine manure effluents using freshwater algae: fatty acid content and composition of algal biomass at different manure loading rates. *Journal of applied phycology*, 20(6):1079–1085, 2008.

[195] L. Mussi, F. Daoilo, and S. Cagoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, 181:4642–4657, 2011.

[196] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[197] F. Neumann and C. Witt. *Bioinspired Computation in Combinatorial Optimization*. Springer, 2010.

[198] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[199] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 2006.

[200] M. J. North, C. M. Macal, J. S. Aubin, P. Thimmapuram, M. Bragen, J. Hahn, J. Karr, N. Brigham, M. E. Lacy, and D. Hampton. Multiscale agent-based consumer market modeling. *Complexity*, 15(5):37–47, 2010.

[201] NVIDIA. *Parallel Thread Execution*, ISA version 3.1 edition, September 2012.

[202] NVIDIA. *CUDA C Programming Guide*, 5.0 edition, July 2013.

[203] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with cuda. In H. Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[204] R. C. S. Oliveira, C. Esperança, and A. Oliveira. Exploiting space and time coherence in grid-based sorting. In A. Cuadros-Vargas and E. J. Y. C. Cahuina, editors, *Workshops of SIBGRAPI (2013)*, pages 55–62, Arequipa, Peru, 2013.

[205] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11:339–363, 2010.

[206] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[207] J. Page, R. Poli, and W. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In *Genetic Programming, Proceedings of EuroGP'99, volume 1598 of LNCS*, pages 39–49, 1999.

[208] S. Palit, S. N. Sinha, M. A. Molla, A. Khanra, and M. Kule. A cryptanalytic attack on the knapsack cryptosystem using binary firefly algorithm. *Int. Conf. on Computer and Communication Technology (ICCCT)*, 2:428–432, 2011.

[209] M. Pall. The luajit project, 2008.

[210] Panait and Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11:387–434, 2005.

[211] S. Papaček, C. Matonoha, V. Stumbauer, and D. Stys. Modelling and simulation of photosynthetic microorganism growth: random walk vs. finite difference method. *Mathematics and Computers in Simulation*, 82:2022–2032, 2012.

[212] M. T. Parker, M. J. North, N. Collier, T. Howe, and J. Vos. Agent-based meta-models. Technical report, Argonne National Laboratory (ANL), 2006.

[213] K. Parsopoulos, V. Plagianakos, G. Magoulas, and M. Vrahatis. Improving the particle swarm optimizer by function "stretching". In N. Hadjisavvas and P. M. Pardalos, editors, *Advances in Convex Analysis and Global Optimization: Honoring the Memory of C. Caratheodory*. Kluwer Academic Publishers, Netherlands, 2001. ISBN 0-7923-6942-4.

[214] K. E. Parsopoulos and M. N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural Computing*, 1:235–306, 2002.

[215] H. V. D. Parunak, R. Savit, and R. L. Riolo. Agent-based modeling vs equation-based modeling: A case study and users' guide. In *Proc. First Int. Workshop on Multi-Agent Systems and Agent-Based Simulation*, number 1534 in LNAI, pages 10–25, Paris, France, 4-6 July 1998.

[216] M. E. H. Pedersen. Good parameters for differential evolution, hl1002. Technical report, Hvass Laboratories, 2010.

[217] M. E. H. Pedersen. *Tuning & Simplifying Heuristical Optimization*. PhD thesis, University of Southampton, 2010.

[218] M. E. H. Pedersen and A. J. Chipperfield. Simplifying particle swarm optimization. *Applied Soft Computing*, 10:618–628, 2009.

[219] M. Pelikán, V. Kvasnicka, J. Pospichal, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo. Reads linear codes and genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, page 268, 1997.

[220] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 116–123, New York, NY, USA, 2008. ACM.

[221] D. P. Playne. *Generative Programming Methods for Parallel Partial Differential Field Equation Solvers*. PhD thesis, Massey University, 2011.

[222] D. P. Playne, M. G. B. Johnson, and K. A. Hawick. Benchmarking GPU Devices with N-Body Simulations. In *Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA.*, pages 150–156, Las Vegas, USA, 13-16 July 2009. WorldComp.

[223] R. Poli and S. Cagnoni. Genetic programming with user-driven selection: Experiments on the evolution of algorithms for image enhancement. In *Genetic Programming 1997: Proceedings of the 2nd Annual Conference*, pages 269–277. Morgan Kaufmann, 1997.

[224] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1:33–57, 2007.

[225] R. Poli, W. Langdon, and N. McPhee. *A field guide to genetic programming*. lulu.com, 2008.

[226] R. Poli and N. F. McPhee. Exact schema theory for GP and variable-length gas with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

[227] R. Poli, L. Vanneschi, W. B. Langdon, and N. F. McPhee. Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11:285–320, 2010.

[228] V. R. Pratt. Top down operator precedence. In *1973 Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973.

[229] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes*. Cambridge University Press, 2007.

[230] M. Privošnik. Evolutionary optimization of emergent phenomena in multi-agent systems using heuristic approach for fitness evaluation. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 1829–1834, May 2009.

[231] M. Privošnik, M. Marolt, A. Kavčič, and S. Divjak. Construction of cooperative behavior in multi-agent systems. In *Proceedings of the 2nd International Conference on Simulation, Modeling and optimization (ICOSMO 2002)*, pages 1451–1453, Skiathos, Greece, 2002. World Scientific and Engineering Academy and Society.

[232] D. Q. Quach, D. P. Playne, and K. A. Hawick. High performance simulations of complex food webs in the Lotka-Volterra predator-prey model. Technical Report CSTN-238, Computer Science, Massey University, Auckland, New Zealand, 2013. INMS PGConf, 2013.

[233] S. F. Railsback and V. Grimm. *Agent-Based and Individual-Based Modeling: A practical introduction*. Princeton University Press, 2012.

[234] S. F. Railsback, S. L. Lytinen, and S. K. Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(609):609–623, 2006.

[235] R. Ranjbar, R. Inoue, H. Shiraishi, T. Katsuda, and S. Katoh. High efficiency production of astaxanthin by autotrophic cultivation of *Haematococcus pluvialis* in a bubble column photobioreactor. *Biochemical Engineering Journal*, 39:575–580, 2008.

[236] A. S. Rao and M. P. Georgeff. Modeling rational agents within a bdi-architecture. Technical report, Australian Artificial Intelligence Institute, 1991.

[237] A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, 1995.

[238] K. Reinholtz. Java will be faster than c++. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, California 91109-8099, 2000.

[239] M. Resnick. *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. MIT Press, 1994.

[240] C. Reynolds. Flocks, herds and schools: A distributed behavioral model. In Maureen C. Stone, editor, *SIGRAPH '87: Proc. 14th Annual Conf. on Computer Graphics and Interactive Techniques*, pages 25–34. ACM, 1987. ISBN 0-89791-227-6.

[241] C. Reynolds. Boids background and update. Accessed at http://www.red3d.com/cwr/boids., July 2011.

[242] T. J. Richer. The Lévy particle swarm. In *IEEE Congress on Evolutionary Computation*, 2006.

[243] A. Richmond and J. U. Grobbelaar. Factors affecting the output rate of *Spirulina platensis* with reference to mass cultivation. *Biomass*, 10(4):253–264, 1986.

[244] P. Richmond. FLAME GPU technical report and user guide. Department of Computer Science Technical Report CS-11-03, University of Sheffield, 2011.

[245] P. Richmond, S. Coakley, and D. M. Romano. A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1125–1126, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[246] P. Richmond and D. Romano. Agent Based GPU, a real-time 3D simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings International Workshop on Super Visualisation (IWSV08)*. Press, 2008.

[247] P. Richmond, D. Walker, S. Coakley, and D. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, 11(3):334–347, 2010.

[248] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3:175–184, 1960.

[249] C. Ryan, J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, April 1998. Springer-Verlag.

[250] L. B. Said, T. Bouron, and A. Drogoul. Agent-based interaction analysis of consumer behaviour. In *Proceedings of the First International Joint Conference on Autonomous agents and multiagent systems: part 1*, pages 184–190. ACM, July 2002.

[251] Sayadi, Ramezanian, and Ghaffari-Nasab. A discrete firefly meta-heuristic with local search for makespan minimization in permutation flow shop scheduling problems. *Int. J. of Industrial Engineering Computations*, 1, 2010.

[252] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Ecological Modelling*, 80:161–170, 1995.

[253] T. C. Schelling. Dynamic models of segregation. *Journal of Mathematical Sociology*, 1(2):143–186, 1971.

[254] M. Scheutz and P. Schermerhorn. Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. In *Journal of Parallel and Distributed Computing*, 2006.

[255] C. Schulz, G. Hasle, A. R. Brodtkorb, and T. R. Hagen. GPU computing in discrete optimization. part ii: Survey focused on routing problems. *Euro J. Transp. Logist.*, Online:1–26, 24 April 2013.

[256] J. F. Schutte, B. J. Fregly, R. T. Haftka, and A. D. George. A parallel particle swarm optimizer. Technical report, University of Florida, Department of Electrical and Computer Engineering, Gainesville, FL, 32611, 2003.

[257] C. J. Scogings and K. A. Hawick. Altruism amongst spatial predator-prey animats. In S. Bullock, J. Noble, R. Watson, and M. Bedau, editors, *Proc. 11th Int. Conf. on the Simulation and Synthesis of Living Systems (ALife XI)*, pages 537–544, Winchester, UK, 5-8 August 2008. MIT Press.

[258] C. J. Scogings and K. A. Hawick. An Agent-Based Model of the Battle of Isandlwana. In *Proc. 2012 Winter Simulation Conference*, number CSTN-116, Berlin, Germany, 9-12 December 2012. WSC. ISBN: 978-1-4673-4780-8.

[259] J. Senthilnath, S. N. Omkar, and V. Mani. Clustering using firefly algorithm: Performance study. *Swarm and Evolutionary Computation*, 2011.

[260] D. Servat and A. Drogoul. Combining amorphous computing and reactive agent-based systems: a paradigm for pervasive intelligence? In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 441–448. ACM, 2002.

[261] G. P. Settembre, P. Scerri, A. Farinelli, K. Sycara, and D. Nardi. A decentralized approach to cooperative situation assessment in multi-robot systems. In Padgham and Parkes and Muller and Parsons, editor, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 31–38, Estoril, Portugal, May 2008. ISBN:978-0-9817381-0-9.

[262] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings*, 1998.

[263] M. Sierhuis, W. J. Clancey, and R. van Hoof. Brahms: A multi-agent programming language for simulating work practice. Technical report, RIACS/NASA Ames Research Center, 1999.

[264] M. Sierhuis, W. J. Clancey, and R. van Hoof. Brahms: a multiagent modeling environment for simulating social phenomena. In *First Conference of the European Social Simulation Association (SIMSOC VI)*, Groningen, The Netherlands, 2003.

[265] J. K. Singer. Parallel implementation of the fast multipole method with periodic boundary conditions. *East-West J. Numer. Maths*, 3:199–216, 1995.

[266] T. Soule and J. A. Foster. Code size and depth flows in genetic programming. In J. Koza, K. Deb, , M. Dorigo, D. Fogel, M. Garzon, and H. Iba, editors, *Proceedings of the 2nd Annual Conference on Genetic Programming*, pages 313–320. MIT Press, 1997.

[267] J. C. Spall. *Introduction to Stochastic Search and Optimization*. Series in Discrete Mathematics and Optimization.

Wiley-Interscience, 2003.

[268] G. Spezzano and D. Talia. Designing parallel models of soil contamination by the CARPET language. *Future Generation Computer Systems*, 13(4-5):291–302, 1998.

[269] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56:91–99, 2001.

[270] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[271] F. Stonedahl and U. Wilensky. Finding forms of flocking: Evolutionary search in abm parameter-spaces. In *Multi-Agent-Based Simulation XI*. Springer, 2011.

[272] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[273] W. Taha. Domain-specific languages. In *Pro. Int. Conf. Computer Engineering and Systems (ICCES)*, pages xxiii – xxviii, 25-27 November 2008.

[274] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.

[275] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248:211–242, 2000.

[276] Terna. Simulation tools for social scientists: Building agent based models with SWARM. *Journal of Artificial Societies and Social Simulation*, 1(2), 1998.

[277] S. R. Thangiah, O. Shmygelsaka, and W. Mennell. An agent architecture for vehicle routing problems. In *Proc 2001 ACM Symposium on Applied Computing*, pages 517–521, Las Vegas, USA, 2001. ISBN 1-58113-287-5.

[278] J. C. Thiele and V. Grimm. NetLogo meets R: Linking agent-based models with a toolbox for their analysis. *Environmental Modelling & Software*, 25:972–974, 2010.

[279] Tisue and Wilensky. NetLogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, 2004.

[280] J. Togelius, R. D. Nardi, and A. Moraglio. Geometric PSO + GP = Particle Swarm Programming. In *2008 IEEE Congress on Evolutionary computation (CEC 2008)*, 2008.

[281] C. Ugwu, H. Aoyagi, and H. Uchiyama. Photobioreactors for mass cultivation of algae. *Bioresource technology*, 99(10):4021–4028, 2008.

[282] K. Usbeck and J. Beal. An agent framework for agent societies. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 201–212. ACM, 2011.

[283] S. van Berkel. Automatic discovery of distributed algorithms for large-scale systems. Master's thesis, Delft University of Technology, 2012.

[284] S. van Berkel, D. Turi, A. Pruteanu, and S. Dulman. Automatic discovery of algorithms for multi-agent systems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 337–334, July 2012.

[285] F. van den Bergh. *An Analysis of Particle Swarm Optimisers*. PhD thesis, Department of Computer Science, University of Pretoria, Pretoria, South Africa, 2002.

[286] Venter and Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *6th World Congresses of Structural and Multidisciplinary Optimization*, 6, 2005.

[287] A. Voicu and C. Galalae. Large-scale agent-based models in marketing research: the quest for the mythical free lunch. *International Journal of Business and Economics Research*, 2(3):33–40, 2013.

[288] V. Volterra. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Mem. R. Accad. Naz. dei Lincei, Ser VI*, 2, 1926.

[289] M. Walker. Introduction to genetic programming. Downloaded from http://www.cs.montana.edu/ bwall/cs580/introduction_to_gp.pdf February 5, 2013.

[290] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proc. ACM IEEE Conf on Supercomputing (SC92)*, 1992.

[291] M. S. Warren and J. K. Salmon. A parallel hashed Oct-tree n-body algorithm. In *Supercomputing*, pages 12–21, 1993.

[292] T. Weise. *Global Optimization Algorithms - Theory and Application*. Self-Published, second edition, June 26, 2009. Online available at http://www.it-weise.de/.

[293] T. Weise, M. Zapf, R. Chiong, and A. J. Nebro. Why is optimization difficult? In *Nature-Inspired Algorithms for Optimisation, SCI 193*. Springer-Verlag, 2009.

[294] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.

[295] M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[296] X. Wu and J. C. Merchuk. A model integrating fluid dynamics in photosynthesis and photoinhibition processes. *Chemical Engineering Science*, 56(11):3527–3538, February 2001.

[297] H.-R. Yang, K.-K. Kang, and D. Kim. Acceleration of massive particle data visualization based on gpu. In *Virtual and Mixed Reality, Part II HCII 2011*, number 6774 in LNCS, pages 425–431. Springer, 2011.

[298] X.-S. Yang. *Nature-Inspired Metaheuristic Algorithms*. Frome: Luniver Press, 2008.

[299] X.-S. Yang. Firefly algorithms for multimodal optimization. In *Stochastic Algorithms: Foundations and Applications, SAGA*, 2009.

[300] X.-S. Yang. Firefly algorithm, Lévy flights and global optimization. In M. Bramer, R. Ellis, and M. Petridis, editors, *Research and Development in Intelligent Systems XXVI*, pages 209–218. Springer London, 2010.

[301] X.-S. Yang. Test problems in optimization. In X.-S. Yang, editor, *Engineering Optimization: An Introduction with Metaheuristic Applications*. John Wiley & Sons, 2010.

[302] X.-S. Yang, S. S. S. Hosseini, and A. H. Gandomi. Firefly algorithm for solving non-convex economic dispatch problems with valve loading effect. *Applied Soft Computing*, 12:1180–1186, 2012.

[303] Y. Zhang, L. Liu, X. Lu, and D. Li. Efficient range query processing over DHTs based on the balanced Kautz tree. *Concurrency and Computation: Practice and Experience*, 23:796–814, 2011.

[304] Zhou and Tan. GPU-based parallel particle swarm optimization. *Evolutionary Computing*, 2009.

[305] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson. Evolving accurate and compact classification rules with gene expression programming. *IEEE Transactions on Evolutionary Computation*, 7:519–531, December 2003.

APPENDIX A LIST OF PUBLICATIONS

The following bibliography is a list of articles which have been published during the course of this PhD programme.

A. V. Husselmann and K. A. Hawick. 3D vector-field data processing and visualisation on graphical processing units. In *Proc. Int. Conf. Signal and Image Processing (SIP 2012)*, number CSTN-140, pages 92–98, Honolulu, USA, 20-22 August 2012. IASTED

A. V. Husselmann and K. A. Hawick. Parallel parametric optimisation with firefly algorithms on graphical processing units. In *Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM'12)*, number 141 in CSTN, pages 77–83, Las Vegas, USA, 16-19 July 2012. CSREA

A. V. Husselmann and K. A. Hawick. Simulating species interactions and complex emergence in multiple flocks of Boids with GPUs. In T. Gonzalez, editor, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 100–107, Dallas, USA, 14-16 Dec 2011. IASTED

A. V. Husselmann and K. A. Hawick. Spatial agent-based modelling and simulations - a review. Technical Report CSTN-153, Computer Science, Massey University, Albany, North Shore,102-904, Auckland, New Zealand, October 2011. In Proc. IIMS Postgraduate Student Conference, October 2011

A. V. Husselmann and K. A. Hawick. Spatial data structures, sorting and GPU parallelism for situated-agent simulation and visualisation. In *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods (MSV'12)*, pages 14–20, Las Vegas, USA, 16-19 July 2012. CSREA

A. V. Husselmann and K. A. Hawick. Random flights for particle swarm optimisers. In *Proc. 12th IASTED Int. Conf. on Artificial Intelligence and Applications*, Innsbruck, Austria, 11-13 February 2013. IASTED

A. V. Husselmann and K. A. Hawick. Particle swarm-based meta-optimising on graphical processing units. In *Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC 2013)*, Phuket, Thailand, 10-12 April 2013. IASTED

A. V. Husselmann and K. A. Hawick. Levy flights for particle swarm optimisation algorithms on graphical processing units. *Parallel and Cloud Computing*, 2(2):32–40, April 2013

A. V. Husselmann and K. A. Hawick. Genetic programming using the Karva gene expression language on graphical processing units. Technical Report CSTN-171, Computer Science, Massey University, Auckland, New Zealand, July 2013

A. V. Husselmann and K. A. Hawick. Geometric optimisation using karva for graphical processing units. Technical Report CSTN-191, Computer Science, Massey University, Auckland, New Zealand, February 2013

A. V. Husselmann and K. A. Hawick. Visualisation of combinatorial program space and related metrics. Technical Report CSTN-190, Computer Science, Massey University, Auckland, New Zealand, 2013

K. A. Hawick and A. V. Husselmann. Photo-penetration depth growth dependence in an agent-based photobioreactor model. Technical Report CSTN-204, Computer Science, Massey University, Auckland, New Zealand, Las Vegas, USA, July 2013

A. V. Husselmann and K. A. Hawick. Geometric firefly algorithms on graphical processing units. In *Cuckoo Search and Firefly Algorithm*, pages 245–269. Springer, 2014

A. V. Husselmann, K. Hawick, and C. Scogings. Model structure optimisation in lattice-oriented agent-based models. Technical Report CSTN-222, Computer Science, Massey University, 2014. Submitted to the International Journal of Modelling and Simulation (ACTA Press)

A. V. Husselmann and K. Hawick. Intelligent individual agent-based simulation of photobioreactors and growth control. In *Proceedings of the 14th IASTED International Symposium on Intelligent Systems and Control (ISC 2013)*, 2013

A. V. Husselmann and K. A. Hawick. Multi-stage high performance, self-optimising domain-specific language for spatial agent-based models. In *The 13th IASTED International Conference on Artificial Intelligence and Applications*, Innsbruck, Austria, February 2014. IASTED

A. V. Husselmann and K. A. Hawick. Simulating growth kinetics in a data-parallel 3D lattice photobioreactor. *Modelling and Simulation in Engineering*, 2013

GLOSSARY

Agent-based Modelling A modelling methodology which has as its core the principle of generative, “bottom-up” design of macro-level phenomena.

Agent-based Modelling and Simulation Refers to Agent-based Modelling and its simulation.

Domain-Specific Language A programming language intended specifically for one area of application, therefore specialising in it, foregoing more general constructs in order to better serve its target domain.

Emergence A term used to refer to macroscopic patterns which result from seemingly unrelated local interactions. 13, 209

Evolutionary Algorithms Evolutionary Algorithms are stochastic algorithms which operate in a similar manner to Genetic Algorithms where a population of candidates are maintained and new (better) generations are computed regularly using operators inspired by evolution such as mutation, crossover and selection in the simplest case.

Evolutionary Optimisation Optimisation in either parametric or combinatorial space using an algorithm which could be described as evolutionary.

Firefly Algorithm A parametric, metaheuristic search algorithm inspired by flashing behaviour of fireflies.

Gene Expression Programming A sophisticated evolutionary algorithm with several operators which more closely resemble that of the natural analog.

Genetic Algorithm An evolutionary algorithm which traditionally involves the use of simple crossover, mutation and selection to iteratively improve a population of candidates.

Genetic Programming An evolutionary algorithm which searches through “program space”, where candidates are composed of building blocks.

Geometric Firefly Algorithm A new proposed evolutionary algorithm which searches through “program space”, based on the Firefly Algorithm of Xin-She Yang.

Geometric Particle Swarm Optimiser Algorithm A Particle Swarm Optimiser formally generalised to arbitrary search spaces by Moraglio, Chio and Poli.

Multi-agent Systems Normally used in referring to physical teams of robots which communicate in order to accomplish a common goal.

Multi-stage Programming A programming paradigm in which code is assembled for processing data which arrives in stages distinct from one another.

Particle Swarm Optimiser A stochastic population-based parametric optimisation algorithm.

ACRONYMS