Chapter 1

Introduction

1.1 Introduction

In recent years, *Agent-based modelling* (ABM) has emerged as a robust technique for modelling complex, real-world phenomena. ABMs have been developed to simulate a broad range of natural and man-made systems including pedestrian movement, traffic movement, agricultural land use, social interactions, human and animal disease threats, and flocking behaviour in animals just to name a few[1-9]. Agent-based modelling has its origin in artificial intelligence applications where intelligent systems are designed around *intelligent agents* which are elements that perceive their environment through *sensors* and act upon their environment through *actuators*. The agent's behaviour is defined by an *agent function* that takes its input from the agent's sensors and calculates an action to be carried out using the actuators. This concept has been applied to the application of modelling by replicating entire complex phenomena as a system of multiple interacting agents.

Complex ecological phenomena, such as those mentioned in the previous paragraph, are examples of application domains that are potentially computationally complex (from an agent-based modelling perspective). This is partly due to human or biological systems being inherently complex, but partly because they can be easily broken down into objects and agents. For example, systems involving humans, animals, plants, or even cells and their associated processes can be intuitively mapped to a set of agents with defined interactions and behaviours. It then follows that the detail and scope of such systems will naturally require large numbers of complex agents in order to provide a detail-rich simulation. This capability has led to the use of ABMs in a decision-support role for governments and industry, and fostered a demand for models with higher levels of detail within the individual agents and their interactions that provide an even greater scope for experimentation. In addition to the increasing scale and level of details within emerging agent-based modelling systems, there has been a push (not just with ABMs) to make use of stochastic processing and Monte-Carlo sampling in order to capture the inherent randomness within the ecological and biological systems being modelled. The use of this approach within agent-based models adds yet more processing overhead with the generation of high-quality random numbers, but also creates the requirement for carrying out multiple simulation runs in order to gain statistically significant result sets for analysis. The result of this requirement is that the models must be able to be executed within a timely manner.

In order to meet the computational load that these highly detailed modelling systems demand, like their massive scale and the requirement for multiple simulation runs, methods must be developed to provide high performance within the agent-based models. The work in this thesis develops and assesses two different approaches for improving the run-time performance of massive scale agent-based modelling systems. The first approach looks at applying compression and aggregation techniques to agent-state information in order to reduce the memory foot-print and effectively group identical agents into a single representation. The second approach, which was ultimately adopted in the modelling application covered in this thesis, involves applying *Graphics Processing Unit* (GPU) technologies in desktop computers to execute ABMs using the data-parallel processing approach. *GPU* technologies, originally designed for carrying out the arithmetic

processing required to produce complex three-dimensional graphics, have emerged as a high performance platform that is suitable for many general-purpose processing tasks. The development of technologies such as Nvidia's Compute Unified Device Architecture (CUDA)[10] and OpenCL[11] provide the much needed frameworks for harnessing GPU's general-purpose processing capability using a client-server architecture. The GPU programming approach has been successfully applied to agent-based modelling systems, resulting in promising performance gains and enhanced modelling capabilities. This thesis summarises the agent-based simulation approach, highlights the issues and challenges in applying GPU technologies to agent-based simulations, and provides an implementation capable of supporting a national scale modelling and decision support platform for an invasion of the Australian mainland by an invasive pest.

1.1.1 The Old-World Screwworm Fly and Its Threat to Australia

The Old World Screwworm Fly (OWSWF), *Chrysomya bezziana*, is an insect pest that is endemic to the tropical regions of Asia, the Middle East and Africa. The insect reproduces by laying its eggs in open wounds and mucus membranes of warm blooded mammals. Upon the hatching, the OWSWF larvae eat the living flesh of the host animal, causing injury, secondary infections and in extreme cases death. If this pest was introduced to the Australian mainland, it could have a devastating impact on the livestock industries within the northern regions of Australia. To date, the OWSWF has never been able to successfully invade and establish a population on the mainland, however live OWSWF have been trapped on ships in Darwin harbour and on commercial aircraft returning from locations where there are endemic OWSWF populations. In addition, dead OWSWF have been identified on other vessels that have undergone disinfection procedures designed to kill the insect pest[12]. Other possible methods of invasion that have been identified also include introduction of infested animals being traded by islanders throughout Torres Strait and naturally by free-flight or via infected migrating animals. Although it is difficult to explain why the OWSWF has not spread into Australia via these avenues to date, examples from overseas such as incursions into Oman and Iraq[13], and the brief introduction of the closely related New World Screwworm Fly (NWSWF), *Cochliomyia hominivorax*, into Libya[14] through the transport of infested livestock from North America, provide evidences that there is a risk of the insect establishing within Australia.

The OWSWF has a lifecycle that consists of four stages; egg, larval, pupal and adult. This lifecycle, which is illustrated in figure 1.0, takes between three to four weeks to complete. Adult female OWSWF disperse in the process of searching for suitable mates and subsequently in search of suitable host on which to lay their eggs. The successful survival and development of the OWSWF depends upon a range of bio-climatic factors including the ambient temperatures, the availability of vegetation cover, the suitability of soil for pupation and the availability of hosts with suitable wounds for ovipositing. Based upon analysis and studies of OWSWF biology in South-East Asia, it is believed that significant areas of the Australian mainland, particularly those areas with large livestock populations, may provide appropriate conditions for an OWSWF population to be successfully introduced and established[15].

In the last two decades there have been numerous studies on the potential economic impacts of the introduction of the OWSWF into Australia that predict significant direct economic losses resulting from a reduction in the production of beef, dairy and wool industries. In addition to the direct losses from the invasion, secondary impacts on industries related to the livestock production and the cost of control, like the disruption of industries and activities within the infested zones [16] and the cost of containment and eradication measures such as movement controls and the *Sterile Insect Technique* (SIT)(which has proven successful in eliminating the NWSWF from North America[17]) incur a significant cost. Estimations on the overall losses from such an invasion exceed $400 million per annum (1994 dollars) [18] in the worst case scenario.

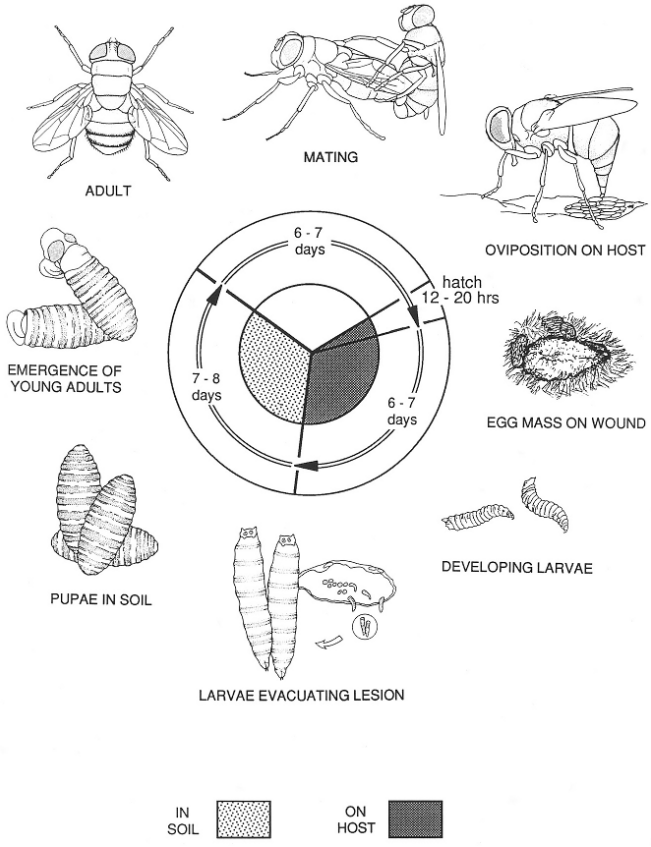

Figure 1.1The lifecycle of the OWSWF and its interactions with host animals. (Source: AHA website)

In response to the threat of the potential introduction of this insect pest into Australia, coupled with the predicted impacts on Australian livestock industries, government authorities have developed a complete readiness plan that is designed to detect, constrain and ultimately eradicate the OWSWF from the continent. This multi-tiered approach, outlined in the AUSVETPLAN[19], calls for the use of a range of measures including surveillance activities to detect OWSWF populations, quarantine of livestock to prevent inadvertent spread and the use of prophylactic treatments and SIT to eradicate the pest.

In order to develop and assess the effectiveness of the preparedness strategies, there is a need for a modelling capability to allow for cost-benefit analysis on a range of scenarios. This modelling capability is not designed to provide a predictive tool in the event of an actual introduction to Australia, but rather a decision support

package that will allow policy makers to model the effectiveness and cost of preparedness strategies to shape policy *before* any introduction to the country. The overarching strategic goal of this modelling capability is to provide a platform that allows for the assessment of the cost and benefits of suitable control measures and develop a sound containment and eradication strategy for any predicted invasion from a range of possible incursion points. To meet this requirement, a hybrid simulation system was formerly developed that used the CLIMEX growth index data layers and the Spatial Analysis System (SPANS) to provide an indirect approximation of the OWSWF spread [20]. As part of the implementation of this earlier system, a population lifecycle-based model was developed to validate spatial distribution and population density results produced by the growth index-based model. Although the population lifecycle-based approach was fully developed from a biological perspective (the economic modelling aspects were never implemented within the system), it was ultimately not included in the final implementation because the growth index-based model was proven to produce an adequate approximation for the OWSWF population dynamics and was far more efficient when running on the desktop computers of the day.

Although this former model has provided valuable insights for policy makers, it has several key short-comings that have limited its utility in responding effectively to actual outbreaks. These include:

- The underlying simulation mechanism for estimating the numbers of female OWSWF is based around an adaptation of the CLIMEX growth modelling package [21]. The simulation system deterministically calculates the number of OWSWF in a location based upon the input data layers. Although this approach is computationally efficient, it fails to capture the underlying biological processes and fails to provide a rich platform for the sensitivity analysis of biological factors that contribute to the lifecycle of the insect.
- The former model makes use of a cellular automata platform to represent the two-dimensional spatial characteristics of the OWSWF's dispersal. Each cell in the grid represents a 20km by 20km area and it assumes that the zone represented within is homogenous in terms of the biological parameters associated with the location. The low resolution model fails to capture the differences in the contributing biological input and potentially over-simplifies the simulation of the invasion.
- The former model uses a deterministic template-based estimation for simulating the dispersal. This approach might not capture well the randomness of the flies' dispersal. In an ideal model, each individual insect would be dispersed according to random samples from appropriate distributions for both the direction and the distance components of the dispersal.
- Spatial Analysis System (SPANS) does not provide the ability to perform spatial queries on the simulation output and the combination of this data with other spatial data sources. This limits the decision support capabilities supplied by the modelling software.

Effective solutions to these short-comings require both innovations in the granularity of the simulation as well as the computational capability to simulate the population lifecycle of each insect in the simulation. Fortunately, since the development of the CLIMEX-based model, there have been significant advancements in computing capabilities and modelling methodologies that have allowed the development of more

sophisticated simulations that use stochastic sampling to capture the probabilistic nature of biological processes, and provide an ability to more directly link the biological processes to the logic implemented in the simulation, in place of weighted estimations such as those implemented in the CLIMEX-based OWSWF model.

The work in this thesis explores the application of agent-based modelling to remedy the short-comings of the CLIMEX-based OWSWF bio-climatic model to produce a modelling system that is capable of capturing the low-level bio-climatic relationships that affect the lifecycle of the OWSWF. This work does not extend to the implementation of the control and eradication measures. Nor does the final implementation include the economic modelling components that capture costs of the invasion. The work here focuses on the implementation of the biological model for the OWSWF, while providing a framework for the additional functionalities to be built in once the initial base biological model is in place.

1.1.2 Methods for Improving ABM Performance

In order to implement the complex bio-climatic model proposed in section 1.1, this thesis explores two approaches for providing a high-performance framework for the development of the insect-level model. The first approach involves the use of data compression and aggregation methods within agent-based simulations. To date, there has only been limited work on the application of compression techniques to agent-based modelling for the purpose of improving model performance. As a result of the limited literature and the lack of understanding of the effectiveness of this approach, a detailed study involving the implementation of a proof-of-concept implementation was undertaken. Within this study, a hierarchical approach for managing agent state information is developed for use with an existing agent-based modelling system. The modelling system used for this case study was the prototype *National Model for Emerging Livestock Disease threats*[22]. This model is designed to simulate an outbreak of infectious disease within cattle, sheep and pig populations within Australia. It draws upon a number of data sources, including climatic and animal movement data, and simulates the spread of disease at the animal level (i.e. each animal is an agent within the system). As such, this simulation provided a good platform to evaluate the effectiveness of applying data compression techniques to the agent-based modelling approach. However, as this thesis discovered, the data compression approach fails to deliver performance gains as anticipated when applied to the OWSWF study due mainly to the huge variations of the agent state information during simulation. This led to the research and development of a second approach for performance improvement based on the GPU technologies.

As mentioned above, the second approach explored in this thesis involves the use of Graphics Processing Unit (GPU) technologies for executing ABMs. This approach has been well covered within the existing literature across a range of modelling systems and provides large performance gains. The key disadvantages of this approach are that the additional GPU hardware is required to run the simulation and additional skill sets are required for GPU programming.

1.1.3 Geographical Information Systems and Agent Based Modelling

Geographical Information Systems (GISs) are software systems that provide the tools to organise and analyse spatial data. They provide researchers with the tools to analyse the relationships between representations of real-world entities based upon their spatial properties. GISs can be used to provide powerful decision support capabilities to agent-based modelling systems by allowing the spatial aspects of the model to be combined with additional spatial datasets from outside the model for a complete analysis. The key challenges involved in the use of GIS with agent-based models include the representation of the spatial aspects of the agent-based model within the GIS and the development of an architecture that allows the spatial information from the ABM to be accessed by the GIS. Another main contribution of this thesis concerns the use of GIS in agent-based modelling and addresses the challenges through the development of a *loosely-coupled* architecture, which allows the capabilities of the GIS to be appropriately applied to the agent-based modelling system.

1.2 Overview

The following chapters include a detailed review of literature related to agent-based modelling for biological phenomena, the development of a hierarchical approach for improving the efficiency of agent-based modelling systems, an implementation of a high-resolution, agent-based model for an OWSWF invasion of Australia that uses the NVidia CUDA GPU platform, a detailed performance analysis and validation of the model's output and finally an outline for an approach for coupling agent-based models with geographical information systems.

Chapter 2

Literature Review

2.1 Agent and Agent-Based Modelling

As mentioned in chapter one, agent-based modelling draws its basis from the intelligent agent systems applied in artificial intelligence applications. They are designed around the principle of conceptually breaking complex systems down into individual components referred to as *Agents.* Although there is no strict definition for what actually constitutes an agent within an ABM, the models and artificial intelligence (AI) systems outlined in literature [23] contain agents that are self-contained and autonomous, who are responsible for managing their own interactions, states and data. These agents interact and update their state in discrete time steps within the simulation and perceive the environment in which they exist. Agents can be heterogeneous in nature within a simulation; a single model can contain limitless 'types' or classes of agents. The systems developed in [24-29] all specify multiple types of agents that interact to simulate phenomena. Agents can be conceptually mobile or static within a simulation. The STREETS simulation outlined in [1] provides an example of agents with different levels of mobility. The model simulates the movement of pedestrians through urban areas by the interaction of mobile 'pedestrian' agents with static 'environment' agents that represent the streets and pathways.

Agent-based models simulate the behaviour of complex systems from the bottom up through the state changes, actions and interactions of individual agents that make up the systems. This is referred to as *Emergent behaviour;* the combined behaviours of the system's component-agents define the behaviour of the system as a whole. A good way to describe emergent behaviour is with the analogy of the cells within organisms. An organism is made up of cells and the behaviour of these cells determines the behaviour of the whole organism in the same way as the behaviour of agents in an ABM determines its overall behaviour. An example of another computational system that demonstrates emergent behaviour is neural networks [23] used in many artificial intelligence applications. In a neural network, the combined action of all the nodes determines the output of the network. Figure 2.1 outlines the general principle of emergent behaviour. In this schematic figure, a heterogeneous set of agents interact amongst themselves while reacting to input from a changing environment. The agents produce individual behaviours within the agent-based system and the system as a whole exhibits the combined emergent behaviour of the external environment. An excellent example of the power of emergent behaviour at different levels can be seen in the SWARM simulation tool kit developed by Minar et al. [29] where agents can be organised into groups (or swarms). These swarms can then be treated as a single agent in the simulation with behaviour defined by the constituent agents.

The key advantages of the agent-based modelling approach centre on the flexibility to model systems at different levels of detail through the emergent behaviour of the agents. By modelling with different levels of detail, scientists can develop rich modelling environments that are suited to decision support and experimentation. Decision makers can use ABMs to experiment with individual-level parameter values and see the effects of their changes on the whole system through emergent behaviour. Agent-based modelling provides a natural way to simulate many real world systems. It is far easier to break a problem down into its component parts and concentrate on modelling each component, rather than develop systems of equations for

modelling a phenomenon as a whole. Another aspect of this mapping to real world concepts is the implementation within programming languages. Agent-based models can be efficiently and easily implemented within object oriented programming languages like Java or C++ without the need for complex mappings of conceptual agents to program procedures and sets of variables.

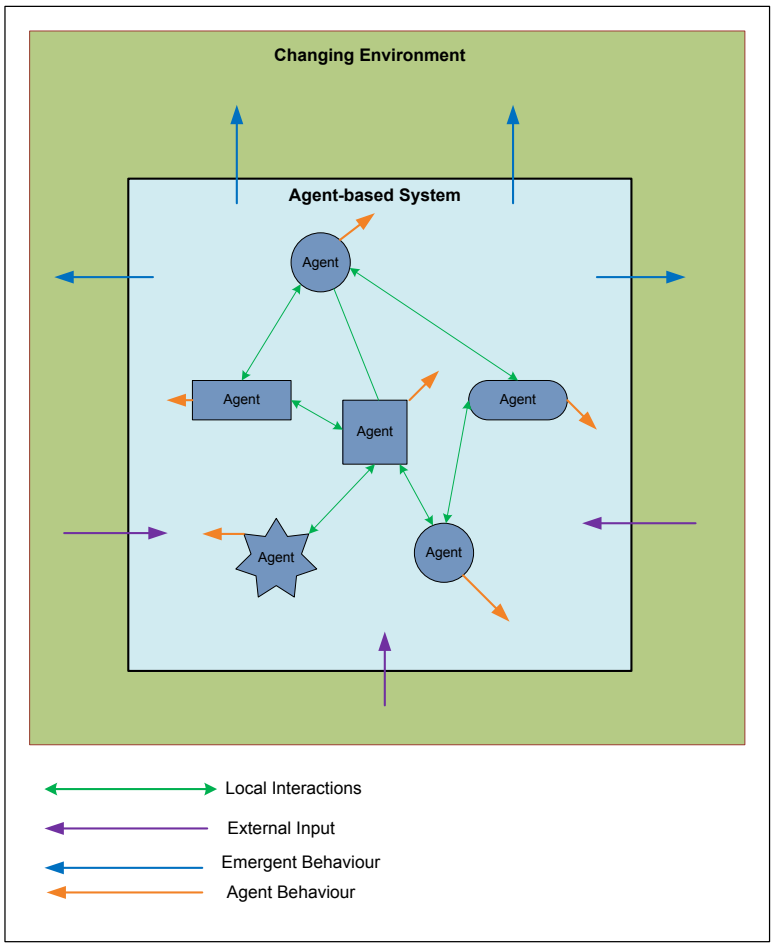

Figure 2.1 Emergent behaviour in a complex system (Source: Welch et.al. [30])

The main disadvantage with this type of modelling, as summarised by Crooks et al. [29], also lies within the high level of details possible within the simulation. Because ABMs use a bottom-up modelling paradigm, the behaviour exhibited by an ABM is only as accurate as the behaviour exhibited by the individual agents within the ABM. Consequently, if the actions of the agents at the lowest level of the simulation are not realistic, then the emergent behaviour exhibited by the system as a whole will not be realistic as well. Obtaining accurate data related to the interactions and states of individual agents within a simulation can be an extremely large undertaking and in some situations impossible. This can make agent-based models expensive to produce and in situations where data is unavailable or difficult to quantify, the accuracy of an ABM can be significantly limited.

The construction of an agent-based model can be a complicated undertaking. The nature of the modelling paradigm allows the development of models that are detailed and flexible but also complicated due to the data requirements and interactions between agents. Crooks et al. addresses the challenges of agent-based

modelling in [30] by dividing them into six key categories:

1. *The Purpose of the model.* This refers to the phenomena that will be simulated and the ultimate role that the model will serve. It is important to have a clear understanding of these as they determine the level of detail in the model and therefore the data requirements and overall the complexity of the simulation.

2. *Theory and Model.* The agent based model needs to be based upon algorithms and theories that can be tested, improved and tuned to accurately simulate the phenomena.

3. *Replication and Experimentation.* The model must be transparent and reproducible by others. This also involves testing the model under different situations to prove that its underlying theories are correct.

4. *Verification and Validation.* The model needs to be tested against other models and existing data sets to insure that it provides an accurate simulation. This can be very difficult with predictive models where confirmatory data sets are unavailable.

5. *Making models operational.* This involves developing and tuning the model so that it can be used to produce output for its intended purpose. This can be made easier by using various agent-based modelling toolkits to produce modules or even entire models. Some of these toolkits are covered in a later section.

6. *Sharing and deploying the model.* The final challenge, which is applicable to other type of models as well, refers to the method of delivering the model's output to its intended users. GIS visualisation packages that allow layers of custom information to be displayed over geographic maps now provide intuitive interfaces that are user friendly for end users.

These challenges must be addressed when developing agent-based models.

2.2 Current Agent Based Modelling Toolkits

Agent based modelling has come to be widely explored as a means to simulate various complex systems in the areas of supply chain management, biology, social sciences and engineering. There are two main engineering approaches that scientists have taken to develop agent based models for their fields. The first has been to develop the software to model the specific complex system being researched from scratch. The second approach involves using an existing framework (and associated software libraries) to implement the model, adding to the software provided by the simulation package to capture the logic and aspects being modelled. There are many software packages (some of them open source and free to use) that are designed to allow users to easily implement models for a range of different complex systems. In the following sections, some of the more popular general agent-based modelling toolboxes will be reviewed followed by a selection of domain-specific toolboxes which are geared for developing simulations of specific types of systems such

as social science or biology. The aim of this section is to provide an overview of existing modelling packages that can provide the features performance for implementing the national agent-based modelling system for OWSWF. In addition to the implementation concerns, the review of these systems also provides an insight into the architecture and organisation of agents and how they are used to model behaviour.

In addition to the review of agent-based modelling systems, the *Cellular Automata* modelling approach is reviewed as a sub-category of agent-based modelling to provide an introduction to the more traditional methods for spatial modelling.

2.2.1 General Agent-based Modelling Toolkits

One of the most mature frameworks in use is the Swarm simulation system outlined in [29]. This work provides a conceptual framework and a set of libraries implemented in objective C that scientists can use and develop additions to build domain specific agent-based simulations. The swarm system has 'agents' as the basic unit within the simulation. Like most agent-based models, an agent is any actor that is involved in events within the model. These agents interact via discrete events, which are scheduled by the system according to a data structure developed as part of the simulation. Therefore, the passage of time is modelled by the sequence of events in the schedule.

The agents in a swarm simulation are organised into 'swarms', with a schedule of events defined for the agents within the swarm. In a simple simulation, a single swarm can constitute the entire model. Swarms are not only containers for other agents, swarms can be agents themselves. In this situation, the behaviour of the agent is defined by the emergent behaviour of the agents it contains. This architecture allows the development of models at different levels, allowing different levels of detail to be simulated. Another feature that sets swarm aside from other tools for simulation is how the system deals with the 'environment' in which the agents exist. Many other modelling systems have fixed environments, for example two dimensional grids that represent geographical locations. In a swarm, the environment is implemented as just another agent. This removes unnecessary complexity from the system. The final distinguishing feature of the swarm system is the way data is collected from the simulation. In the swarm system, 'observer' agents interact with the other agents of the model to collect information and output this information to files and databases for analysis after the simulation. By using the observer agents, the model is separated from the data collection process and the design remains consistent throughout.

A good example of a model that has been implemented using the Swarm simulation system is the STREETS model [1]. The STREETS model is designed to simulate the flow of pedestrians through an urban area to study the effects of different street and footpath layouts. In this model pedestrians are agents that are released into an environment of other agents that represent the walk ways, shops and attractions that pedestrians might want to visit. Socio-economic data is used to give the pedestrians an activity schedule which specifies the shops and locations that the pedestrians may wish to visit. They also possess behavioural characteristics that specify the probability that they will deviate from their goal tasks to visit shops of *interest*. The

environment is also modelled as a set of agents where each agent has a *walkability* attribute that determines its attractiveness for use by pedestrians. For example, footpaths have high *walkability* where roads have a low walkability meaning that pedestrians will prefer to move on footpaths over roads.

The simulation operates by releasing pedestrians into the urban world from specified *gate*s. They move through the urban work according to a predetermined schedule of places they need to visit taking the best route possible according to the available walkways and the walkability of the walkways. The model implements various probe agents to collect information from the simulation as it progresses and displays this to a GUI with representation of the urban area.

The NetLogo modelling environment outlined in [31] provides a high level scripting language and a graphical user interface that allows scientists with limited programming experience to create and give instructions to thousands of interacting agents. In the simulation framework, the model consists of 'turtles' that move over and interact with a grid of 'patches'. These agents interact with each other to model phenomena. For example turtles can be used to model molecules, insects, buyers, passengers or practically any other actor from a complex system. Patches can be used to model trees, walls, terrain, cells or any other form of environment modelled in a simulation. The interface for the simulation system is unique because there is no firm distinction between editing the system and using it. Modifications to the model can be carried out while it is running. This means the agents within the model can be examined at any time and altered making models implemented in this system interactive.

NetLogo is implemented in Java, making it platform independent and includes a full suite of data analysis and visualisation tools on all platforms that allow the user to present data from the simulation and export is for use in other applications. In addition, completed simulations can be exported as applets to be used as web-based simulations.

The MASON toolkit presented in [27] provides a single-process, discrete event simulation environment along with modules for visualisation and analysis. MASON does not provide high level tools, like NetLogo, but rather a minimal Java library for programmers to use for implementing their domain specific models. Because MASON is implemented in Java, it is platform independent. MASON is designed with a layered architecture in mind. The base layer consists of a set of custom data structures used by higher layers. The middle layer is the *model* layer which contains classes for specifying the agent's event schedules, and containers for Java objects that represent the agents in the model. These containers associate these objects with locations representing an environment for the model. This layer can be used to develop command line applications, however the final layer, which is the *visualisation* layer, provides interface and visualisation tools that allow the development of fully graphical simulations. MASON provides objects similar to Swarm's probes, referred to as *field portrayals*, that collect information from the agents as the model runs and passes them to the visualisation layer. The use of layers serves the same purpose as the probe agents in Swarm: the visualisation and analysis tasks are kept separate from the actual model.

The JAS agent-based simulation reviewed in [26] has a very similar architecture to MASON but also provides a protocol for publishing interface components of simulation to the world wide web which allows users to interact with the simulation via a webpage.

The RePast modelling package developed in [32] offers a similar type of framework to that of the Swarm system. RePast is implemented in Java and (like most other toolkits) provides a set of classes for programmers to run, display and collect data from agent-based models. The main design difference between RePast and Swarm is with the scheduling of events. Like in Swarm, the agent's events are scheduled to occur in a specific order. RePast extends this by allowing the model to dynamically schedule events as it runs. This is achieved by allowing the agents events to schedule other events to happen in the future, which allows more complex simulation. The RePast toolkit provides libraries for making use of distributed computing technologies, such as cluster computers, for agent-based modelling. This can allow scientists to quickly develop a high-performance model capable of modelling large scale systems. The example given in [32] proposes a model of a cargo within an airport where a 'flight arriving' event schedules a 'baggage unload' event to occur. It is possible to create simulations where only a single event is explicitly scheduled to start the simulation and all other events are dynamically scheduled from then on.

2.2.2 Domain Specific Agent-Based Modelling Toolkits

The Social Interaction Framework, *SIF*, and its associated Java toolbox presented in [3] provides a system for developing agent-based simulations for the social sciences. The framework is based around the *Effector-Medium-Sensor* paradigm where the model consists of agents that have *effectors* that alter the state of their environment (which is the *medium*) and *Sensors* which perceive the environment (medium). The agents in this system are usually used to represent human beings. The EMS paradigm is implemented in Java through the use of three separate components that each run in their own Java threads. The *World-Server* is responsible for managing the agent's environment and performs administrative tasks such as starting and stopping the simulation and data collection. The *Agents*, which consist of effectors, sensors and different forms of intelligent logic that interact with the environments. The Graphical User Interface, which takes the form of a Java interface, is linked to the simulation as a special type of agent. The GUI uses sensors to receive information from the simulation to display while effectors force agents in the simulation to take specific actions.

When an agent's logic determines it should take an action, the agent activates the appropriate effector. The effector action is picked up by the world server which updates the representation of the environment and computes the input information for all sensors that are affected by the change in the environment. The sensors then perform updates in their agents. This constitutes one cycle for the agents. The actions of one agent may in turn cause other agents to perform action that alter the environment, thus simulating the social complex system.

One of the most complex models reviewed here is the agent based simulation environment for un-manned

combat air vehicle (UCAV) mission planning and execution presented in [24]. This system is designed to simulate the potentially hostile environment that UCAVs are designed to operate in and the UCAVs ability to dynamically modify mission plans in response to changes in its environment. This system provides an agent-rich model where agents are used to represent friendly and enemy military forces that can affect the mission of the UCAV such as enemy aircraft, air defences, ground vehicles etc. The UCAVs themselves are the most significant agents. They are modelled so that they possess an array of different sensors, to perceive their world and detailed decision logic that allows them to alter their plans in response to the environment. From an architectural point of view, the simulation is not that different from the SIF simulation. The *World State Information Collection* component retrieves environment information from raw streams of live data that simulate incoming information from satellites, radar aircraft and ground sources. This data is converted to an environment representation that is perceived by the sensors of the UCAV agents with which they plan mission execution. The sensors of the agents are implemented as functions that operate in their own threads, and constantly wait for updates to the environment that affect them. The UCAV simulation environment includes a user interface that allows the user to specify high level behaviours for the various agents in the simulation and view events that occur throughout the simulation.

Biological systems have received a significant amount of attention in the area of agent-based modelling. The ecosystem simulation presented in [33] is an excellent example of an agent based framework that is tailored for simulating plant and animal interactions in an ecosystem. The model follows a similar structure to that of the Swarm simulation system, being a discrete event driven simulation. However, it includes more specialised features such as an integrated Geographical Information System. The model consists of three main parts:

- The simulator
- The world package
- The user interface

The simulator component contains a synchronisation kernel that schedules events using various helper classes. The world package contains the classes that manage the GIS along with the entities in the simulation. To deal with the unique requirements of simulating the ecosystem, system defines several different classes of entities. *Passive* entities are those objects in the simulation hat do not change during the simulation. These objects have no autonomy and no control is given to them by the synchronisation kernel. Examples of objects represented by passive entities include trees, stones, bowls etc. *Active* entities are those objects which can generate events, but have no memory of their own. These objects are given control flow by the synchronisation kernel to perform tasks during the simulation. *Agents* are objects that can save situations and information into their memory and implement social interactions. Agents are organised into *groups*. Groups assign responsibilities and behaviour to their member agents and facilitate communication between the agents. The example given in [33] clarify how these objects may be used, gives an example of a farm where sheep food is a passive entity, a dog is an active entity, a sheep is and an agent and a flock of sheep is a

group.

The world model interfaces with a GIS to simulate the movement of agents in the model. Three approaches to the management of the locations of agents are offered. The first method, which works best when there are a large number of agents and memory constraints are not an issue, simply involves maintaining a data structure (such as a 2-dimensional array) that records the positions of each object. The second approach, which works best when memory is tight and the CPU time and performance is not an issue, saves location information inside each entity. The final approach is a combination of the first two, where a lower resolution data structure is maintained that specifies the locations of sets of objects and each individual object stores its exact location.

2.2.3 Cellular Automata: A sub-category of Agent-Based Simulation

A *Cellular Automata* (CA) is described in [34] as an array of individual *cells* that each have a set of pre-defined states that they progress through discrete time steps. The new state of each cell at the next time step is determined by a function, called the *local rule*, which takes the state of its neighbours and the current time as inputs. A Cellular Automata traditionally consists of a one or two dimensional array of cells, however with the increasing power of computer technology CAs with higher dimensionality are being developed in various domains. Cellular Automata exhibit emergent behaviour in the same way as agent-based models. The set of cell states at a given time step constitutes the global state of the CA. The global state progresses deterministically over the discrete time steps with application of each cell's local state at each time step. Cellular Automata systems share many similarities with agents-based systems. Each cell in a CA can be viewed as an individual agent with a fixed position in an environment consisting of its neighbouring cells. Agent based systems differ from CA because they can consist of a heterogeneous set of entities, the environment is not uniformly structured and the interactions between the entities can be asynchronous. However, because of the similarities between the agent-based and CA modelling paradigms, many of the design principles applied to CA can be applied to agent-based models and vice-versa.

Cellular Automata were originally devised by Von Neumann in [35] for modelling biological systems such as cell growth and organism evolution. Since this initial development CA has been applied to a range of domains including *DNA Sequence* Modelling [36], path-planning and navigation [37], Electronic Authentication [38] and various simulations that will be looked in later sections. The work in [34] provides a good overview of Cellular Automata. Cellular Automata have four key features, some of which have already been briefly mentioned:

- The geometry of how they represent a system
- The local transition rule for controlling each cell’s state
- The set of states for each cell
- The neighbourhood definition for the cells

The geometry of a CA usually takes the form of an n-dimensional grid. This grid can be infinite, as using the case of many cell growth simulations or a set of boundary conditions can be used to limit the size of the grid. This can be implemented by specifying set of boundary cells that exist in a state that does not change throughout the simulation. Other implementations use a system of mapping functions and predefined positions to represent the geometry. The local rule is usually a deterministic function that controls the transition of a cell's state based upon its own state and the input from cells defined as neighbours. In a traditional CA, each cell has the same local rule, however [34] outlines several implementations where each cell has its own local rule. It is also possible for cells to change their rules over the course of a simulation. Each cell within a CA has its own set of states that is progresses through in accordance with the local rule. It is possible to implement CA where each cell has its own set of states. This type of CA is referred to as *polygeneous*. The neighbourhood of a CA defines the set of cells each individual cell within the CA interacts with. In the original implementation by Von Neumann [35], he used a simple orthogonal neighbourhood for its interactions, however more complex systems have been developed since. It is possible to define a cell's neighbours as a set of input cells and output cell, where the cell's local function accesses the states of its input cells to calculate its next state and makes its state available to its output neighbours for their calculations. Other CAs have been developed where a cell's state is dependent on the sum of its input neighbour states. This is referred to as a totalistic CA.

2.2.4 Generating Complex Simulations by Combining Models

In order to develop diverse, large-scale modelling systems, that combine expertise from different areas, there has been a push to build modelling systems that combine existing modules and even entire simulation systems. Modelling massive systems can require expertise and domain knowledge from a number of different areas. This is due to the interaction of different scientific fields and need to build different areas of a simulation with different levels of detail. This requirement is not unique to agent based models, however here the focus is on this type of model. Simulation frameworks, such as the Swarm simulation system in [29], deal with the need for different levels of detail through the use of grouping agents together and treating these groups as agents themselves, however they do not provide a way to easily plug in additional modules from other simulations. In addition to this, they do not provide the ability to incorporate other existing simulations into the new simulation to create a large, highly detailed model. The ability to combine different simulations and modules, developed independently, allows the development of models that rely on expertise from a range of domains in order to model complex system. Another issue with complex models comes from the homogenous designs that are enforced across all modules in a simulation. Different parts of a model may be more efficiently implemented using different modelling paradigms. This modelling approach is introduced in this review as it could be applied to combine modelling systems already available for simulating OWSWF, livestock movements and other aspects of an OWSWF invasion. The following section analyses the complexities and several implementations for the combination of agent-based modelling systems.

The system presented in [39] aims to support the development of large systems by integrating separately

developed modules. The architecture provides a system that allows agents from different simulation modules to share variables (data) and remain logically synchronised in the simulation. The agents can be developed completely independently and follow their own design paradigms. The simulation environment provides two services: the *time manager*, which insures that all agents progress and remain in the same logical time interval, and the *conflict resolver* which is responsible for managing the access to shared data. The agents in the simulation must contain an interface to these services in order to be incorporated into the simulation.

The time manager dictates the logical time within the simulation. The time progresses based upon requests from the simulation modules. Once a module has finished its processing, it makes a request for the simulation to advance to its next time increment. Once all modules have made their requests, the time manager advances the time to the earliest requested time where the processing cycle begins again. This scheme supports different levels to temporal granularity.

The conflict resolver is responsible for managing access to variables registered as shared amongst multiple agents. The conflict resolver insures that atomic actions are applied consistently as a single transaction and that the combination of internal and external conflict resolution amongst the agents does not result in anomalous failure of actions. For example if a request is made to modify a shared variable at the same logical time by two agents and one of the agents will fail the operation due to other local constraints, this should not block the other agent from executing its action if it can be successful. The example given in [39] illustrates this point with virus treatment. If the variable `antivirus = 1` and the variable `PublicFunding` is only sufficient for one agent with private insurance to receive treatment, a request from two agents at the same logical time where one agent has insurance and one does not should result in the agent with insurance being treated. The successful case should not be blocked by the unsuccessful case just because the requests are issued at the same logical time. In addition to avoid the above situation, the conflict resolver also prevents events that are mutually exclusive from being executed.

The process of adding modules to the simulation involves deciding which variables will be shared and designing the procedures that will interact with the shared variables. The patterns for the procedures are outlined in [39].

The architecture outlined in [39] allowed independent modules that were developed with specified interfaces to be integrated into a single simulation but it does not allow entirely separate, stand-alone, simulations to be combined into a single model. The work outlined in [40] aims to allow the implementation of a set of interacting and co-evolving models that work together to simulate a complex system. This allows reuse of existing models to build up new complex models without the need of direct collaboration during the development or large modifications to the existing models in order to integrate these models. The models to be integrated are treated as black boxes and the only assumptions placed on them are that they have functions that return the current and next simulation time interval. Note that although the focus of this review is on agent-based modelling, this architecture is not specific to agent-based models.

The key challenges involved with integrating the models outlined in [40] include coherency, compatibility

and synthesis. Coherency refers to the way that different simulations represent data. The example given in [40] regards points in a space. Some simulations represent a point using just x and y whereas other may use x, y and z. Compatibility refers to the way individual data items such as integers and floats are stored. Synthesis refers to the way coherency and compatibility are being dealt with. The models that make up the simulation communicate via *coupling-artefacts, Model Agents* and *Model Artefacts*.

The coupling artefacts are responsible for providing the shared data stores between the models. There are coupling artefacts for input and output for each connection (i.e. each integrated models connection to each other model). The coupling artefacts are responsible for the time management such as insuring that data exchanges are carried out in the correct logical order maintaining the causality constraint. The Model Agents are responsible for formatting the data from the model to a universal usable form. This manages the compatibility and coherence issues mentioned above. The Model Agents receive data from the input coupling artefact and output data to the output coupling artefact. The link to the integrated model is complete by the Model Artefact. The Model Artefact interacts directly with the model, accessing the time interface information and the functions responsible for input and output for data. Figure 2.2 illustrates the complete Architecture.

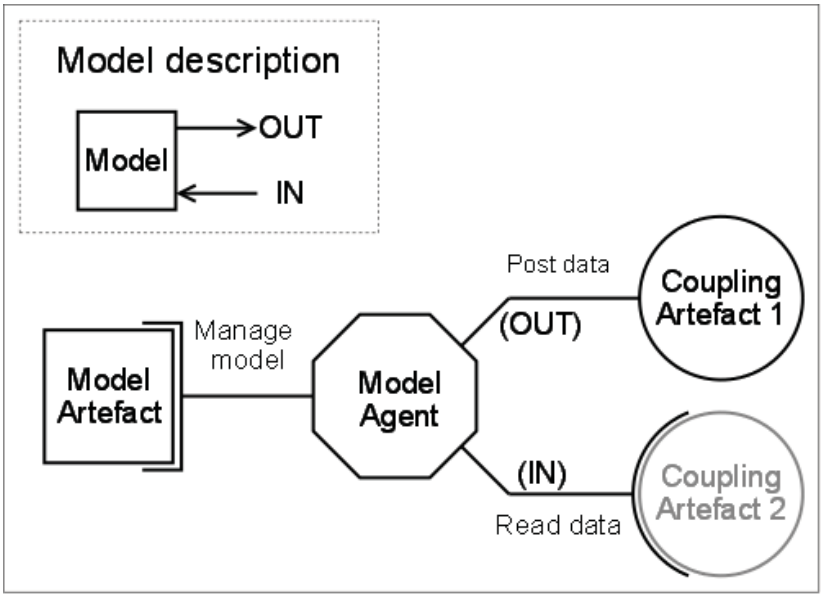

Figure 2.2 The Complete Architecture (Source: [40])

The combination of these three components means that very few modifications are required to the models that make up the overall simulation and it is easy to add, remove or interchange models in the simulation. A proof-of-concept implementation carried out in [40] demonstrates the ability of the architecture to integrate multiple models to simulate sheep herding in an ecosystem.

The United States Department of Defence Modelling and Simulation Office have developed the High Level Architecture (HLA) [41] for developing models that can interact with each other to build larger models. The HLA provides designers and engineers with a set of standard components to base their models around so that they can interact with other models built according to the architecture. The architecture does not specify

programming languages or modelling paradigms that must be followed, but rather specifies interfaces to access shared resources and provides an operating system for the different models to interact. The architecture is designed to support a distributed modelling approach, allowing different simulations to operate on separate devices. Like other frameworks outlined above, the motivation for developing the HLA is the idea that one simulation cannot possibly meet the needs of all users and that interoperation and reuse of simulation can be used to develop simulations that better meet the needs of different users.

The HLA consists of several different components that form a HLA compliant model. The overall model (consisting of one or more other models) is known as a *federation* and the component models to the federation are referred to as *federates*. As mentioned above, the HLA does not specify how information is represented or the modelling paradigms used within the individual federates, but it does require them to include capabilities for functions to communicate with other federates. The component responsible for managing the federation is the *Runtime Infrastructure* (RTI). The RTI provides a set of services (covered in the next section) that manage the interaction, synchronisation and coupling for the federates. The component responsible for communication between the federates and the RTI is the *Interface.* The Interface provides a standard way to access the RTI's services and for the RTI to access the individual federates. The final components specified in the architecture are the *Data Collector,* which is responsible for collating output from the simulation, and the interfaces to *Live Players*, which are simulated instrument panels and or command and control systems that live participants use to act in the simulation. Both of these components communicate via the interface component and as a result are seen by the RTI as standard federates. Figure 2.3 presents an overview of a HLA federation.

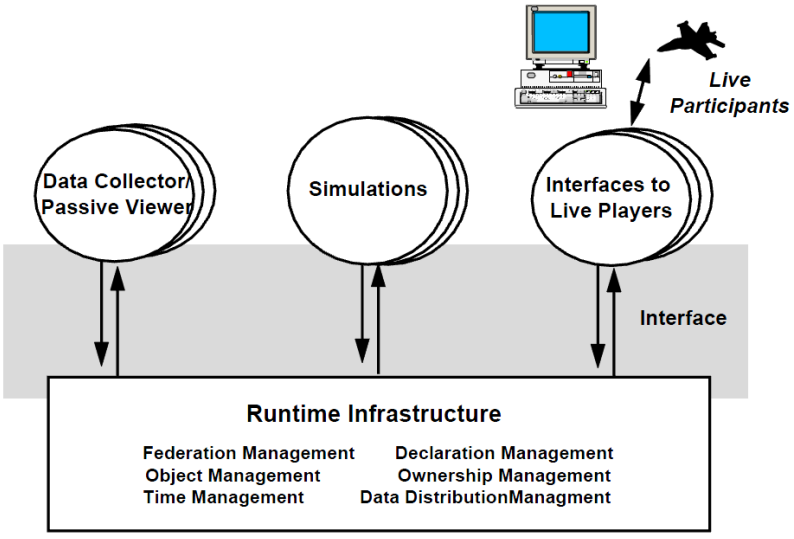

Figure 2.3 Overview of a HLA federation (Source: [42])

The HLA interface specifies the services that the federates can access from the RTI. As figure 2.3 shows, there are six classes of services that are provided by the RTI. *Federation Management* services allow the RTI to create and manage the low level tasks to run the federation. The *Declaration Management* service allows the federates to specify the data that they will make available in the simulation and the data that they will

require access to. The *Object Management* service handles shared objects with the simulation at the object level. The *Ownership Management* service is responsible for the transfer of objects amongst the federates during execution. The *Time Management* service is responsible for co-ordinating the logical time amongst the federates in the simulation and the *Data Distribution* service efficiently routes data amongst the federates over the course of the simulation.

The HLA rules, interface specification and model provided the minimum set of essential tools for interpretability. It insures the basic level of compatibility for information exchange amongst modules and the model for routing data through the federation.

2.3 High-Performance Agent-Based Modelling Approaches

As mentioned in chapter one, there is an increasing push to develop agent-based modelling systems with immense processing requirements. This section reviews high-performance modelling platforms that use technologies such as distributed (cluster) computers, compression technologies and Graphics Processing Unit techniques to provide the computational performance required for executing the ABMs.

2.3.1 Massive Agent-Based Modelling Using Cluster Computers

The simulation frameworks proposed in the previous sections were aimed at allowing simulations with up to thousands of agents interacting in the simulation world. Although this could be seen as a large simulation, *massive* agent based models can contain millions of agents interacting in conceptually large environments. Most of these types of simulations are distributed across multiple processors or systems and are executed in massively parallel ways. A review of work in this area has shown that there have been few attempts to develop models of this scale.

The framework ZASE (*Zillions of agents-based Simulation Framework*) outlined in [42] aims to allow massive agent-based simulations, which can contain millions of agents to be implemented and executed on one or across multiple standard desktop computer systems. The ZASE framework, which is implemented in Java, provides a system to break a simulation down into sets of *simulation runtimes* and *agent runtimes*. The simulation runtimes contain management mechanisms of the simulation environment. The agent runtimes provide the management mechanisms for up to millions of independent, autonomous agents. The framework aims to connect multiple distributed agent processes through the use of message communication, allowing concurrent processing amongst many threads of execution, while allowing agent interaction.

The division of the simulation amongst the runtimes differs based upon the nature of the simulation. The framework does not specify an exact breakdown for all simulation types. The evaluation of the framework was done by using a simulation of an online auction. This simulation is broken down into a single simulation runtime and multiple agent runtimes, where the simulation runtime multicasts bid requests to the agents and the agents respond. The approach used for geospatial simulations will be different as agents will interact with their close neighbours as well as their environment.

The ZASE framework applies a *thread-pool* architecture where agents do not 'own' specific threads and execution of each agent's processing is scheduled amongst the available threads of execution by the runtime that it is part of. Each agent maintains a queue of messages and when a thread becomes available from the pool, the agent is allocated the thread to process the message. Once the processing is done, the thread is allocated to another agent.

Other work into large-scale agent-based models has involved developing distributed systems that partition the work load between several different computer nodes. The AGlobeX simulation platform outlined in [43] uses a dynamic system of partitioning to divide up larger scale agent-based simulation over multiple processing cores. The system divides the simulated environment into partitions based upon the numbers of nodes available and agents present in each area of the simulation. The simulation system also provides dynamic load balancing between the nodes of the simulation as the simulation progresses. Communication is available between agents on nodes, however this is minimised due to the fact that most of an agent's interactions are with its neighbours and the agent's processing is partitioned based on spatial locality.

Each of the agents in the system is broken into three components. The agent's *body* component manages the evolution of the agent's state. The examples given in [43] are the motion dynamics for aircraft in an air-traffic simulation. The *Reactive Control* component handles the agent's states based on the observation of other states. The example given is the control for avoiding collisions. The *Deliberative Control* component contains complex algorithms responsible for long term planning of agent behaviour, communication with other agents and prediction. This component provides the high-level control based on sensing perceptions.

The Architecture of the simulation system consists of Environmental Simulation Components (ESCs) which control the agent's state updaters and the set of *Deliberative Control* modules. Both of these can be separately distributed across the available nodes. Each partition is allocated a single ESC which is responsible for agents that are present within the partition. The Deliberative control modules are distributed across the available nodes, irrespective of the agent to which the module belongs, is situated. A communication channel is maintained that allows an agent's state information and reactive control components to be processed on one node and the deliberative control to be processed on another. Dynamic load balancing is carried out on both levels based upon the required level of processing. Re-organisation of the ESCs on the nodes is only done when the difference between cycle execution times on the nodes surpasses a predefined threshold, rather than every cycle. This is due to the overhead incurred during a reorganisation. Reorganisation of the deliberative components is only done when a new deliberative module is created. They are distributed amongst the nodes based upon the average load over previous simulation cycles.

The model uses a time stepping paradigm, the same as the Swarm-type simulations, where each simulation cycle represents a uniform period of time. The ECSs are responsible for controlling the simulation cycle which is decomposed into several phases. The cycle consists of an update phase, where the state updates that do not require external information are calculated, an evaluation phase, where events that affect multiple

agents are calculated (i.e. agent communication), a migration phase where agents can move from one partition to another if their new location is not within their current ESC, and an application phase where the state updates are applied.

The Distributed Multi Agent System Framework (DMASF) [44] is a system designed to be able to simulate billions of agents in thousands of seconds. Like the other massive agent based systems reviewed here, the DMASF uses distributed computation to achieve the performance required for the simulation. The DMASF has a dynamic scheduler that allocates agents to different nodes, based upon the performance of the nodes. In order to manage the memory requirements of a simulation containing possibly billions of agents, the system uses a database to save and retrieve agent's states throughout the simulation. The result of this is that only a small, fixed, number of agents are kept in the main memory and these are swapped in and out for agent processing. The system is divided into a client-server architecture where a *server* decides which agents are processed on which client machines (i.e. the nodes responsible for the simulation processing) and the synchronisation of the simulation amongst the client nodes.

Benchmark simulations implemented in the DMASF show that it scales linearly with the number of agents models and the processing power dedicated to the simulation. One of the limitations identified in the current implementation concerns the GUI for the simulation framework. Due to communication constraints, mismatches occur between the backend carrying out the simulation and the interface presented to the user. Further refining of this area of the system is required.

The work outlined in [45] implements a parallel, agent-based simulation of the browsing behaviour of users on the World-Wide Web. The hardware that the system is based on consists of a cluster of desktop computers. In the evaluation in [45], a cluster of 16 nodes is used to test the implementation. The simulation is a Swarm-type, discrete time simulation that consists of two basic types of agents: *users*, which represent human users of the web and *websites*. The user agents are linked together in a graph representing a social network and website agent maintains a set of bookmarked websites. The social network is used to model aspects such as discovering new websites by "word-of-mouth". The websites are linked together by a similar graph that represents the hyper links between sites. At each time step within the simulation, the user agents visit website, update their bookmarks and discover website via their social network and the hyperlinks between websites.

The simulation framework provides a modular structure which allows different behaviour to be applied to the simulated users, different graph models for the social and website networks and websites behaviours. Infrastructure component is responsible for running the simulation and producing the output data. The simulation is distributed between the nodes of the simulation based upon the performance of the nodes. Each node is responsible for a set of model users. The simulation uses a simple stochastic economic model that addresses the behaviour of internet browsers. Users are modelled as entities with states, having a short term *loyalty* memory for remembering sites they visit and long term *bookmark* memory. The social network and the World Wide Web are represented by two separate *small-world* graphs with a fixed topology and number

of nodes. Each website has a ranking that represents the attractiveness of the website. In each time step of the simulation a user can carry out the following behaviour:

- Visit bookmarker sites.
- Contact other users and ask them for sites to visit (Word of mouth).
- Surf links from site they have visited this time step.
- Evaluate the attractiveness value of sites they have visited and alter their bookmarks and loyalty entries.

The simulation is implemented using the Oz and Mozart platforms [46]. Oz is a programming language that provides variables that are lexically scoped and various basic data types. The language offers multithreading and synchronisation via *dataflow variables*. Mozart is a run-time environment that provides network-transparency (i.e. the program runs on a cluster of computers as if it were on a single machine). This simulator is broken into several components. The *manager* component is responsible for controlling aspects of the simulator such as time. A *worker* consists of several components that conduct the parallel simulation, each possessing several threads. Each worker includes a *simulator* component which is responsible for executing a user's behaviour using information from the *Users* and *Sites* modules. The users and sites modules hold the network data for the simulated social network and hyperlinks. This data is partitioned amongst the modules but they provide transparent access to all data. Figure 2.4 shows the architecture of the simulation.

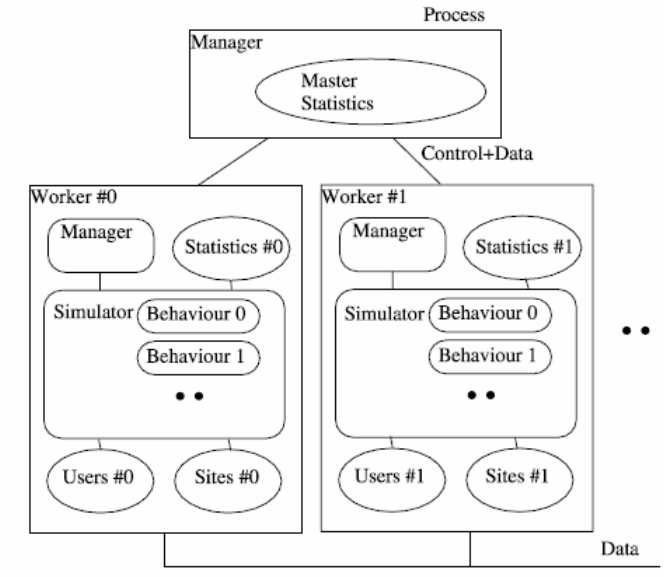

Figure 2.4 The Simulator (Source: [45]).

Each worker component contains a *server* module that allows the data from the users that the worker contains to be accessed by other worker components in the simulation. For example, a user in one worker component can access a list of sites to visit (representing word-of-mouth) from another user contained within another worker component. This is achieved by using a set of functions provided by the worker's server modules. This is outlined in figure 2.5.

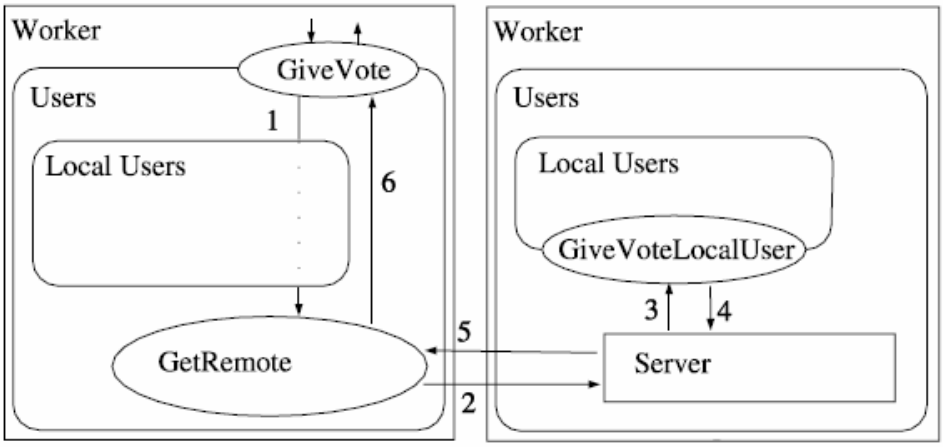

Figure 2.5 Accessing Distributed Data. (Source: [45])

2.4 Applying Data Compression Techniques to Improve Massive Agent-based Model Performance

One way to improve the performance of massive agent-based models is to compress and aggregate agents to different levels of granularity within the simulation. Compression is commonly used to improve the efficiency of applications in other areas of computer science such as in video and audio software. Compression essentially involves finding similarities within data and applying efficient representations of the data [46]. This can either be done in a static fashion before the simulation begins or dynamically as the simulation progresses. The work in [47] presents a system whereby multiple agents are represented by an aggregate agent. The algorithm works by treating each agent as a point in a multi-dimensional space. Each dimension in the agents space represents one of the attributes that the agents possess. Examples of these could include age, location co-ordinates, disease life cycle parameters etc. Clustering algorithms are used to partition all the agents in the space into groups (i.e. clusters) of agents that possess similar attribute values. From here the centroid of each cluster is calculated and the attribute values of this centroid are used to define a single agent that represents all agents in the cluster. In addition to this, the aggregated agent stores an *expansion factor* which indicates the number of agents that the aggregated agent represents. This is a one-off static process carried out at the start of the simulation. There is no alteration of the clusters and aggregate agents over the course of the simulation.

The aggregation method is also explored by Parry and Evens in [48],where populations of agents (representing aphids) are grouped into *super-individuals*. These super individuals statistically represent a group of agents and provide a group the processing for the underlying cohort of individuals. This work showed that the aggregation approach can improve the performance of an agent-based, however when the super agents are used to represent groups as small as 10 agents, the aggregate statistics fail to represent the deviations amongst individuals in the underlying agent population.

Wendel and Dibble build upon this idea with the work presented in [49] where they present a system for dynamically compressing agents, allowing them to move in and out of their compressed state as their internal states change during the course of a simulation. The system works in either a *lossless* mode, where no

information from each agent is lost during compression, or a *lossy* mode, which provides a higher level of compression but results in some information loss during the compression process. The simulation system consists of two components: a set of agent containers that represent the clusters of agents in simulation and a compression manager, which acts as a 'wrapper' around the set of agents by coordinating model interactions with the agent containers. The agent containers contain a count of the number of agents that they represent, just the same as the system outlined in [47]. At the start of the simulation, the compression manager groups agents that are similar into containers. The agents that are particularly unique are left un-grouped. The agent containers behave exactly like the agents that they contain. The compression manager manages the queries to the agents by directing the queries to the appropriate containers or unique agents. The containers respond to the queries just as if they were individual agents. As the internal states of agents change and they differentiate themselves from their groups, the compression manager removes them from the containers and instantiates them as unique agents. If a unique agent's attributes change to match those of the agents present within an existing container, the agent is added to the container.

The method used to group the agents into the containers determines whether the compression is lossy or loss less. If the agents grouped into a container are *identical*, that is all of their attributes have identical values, then the compression will be lossless. If agents that are *similar,* that is their attribute values are similar, are grouped into the same containers then the compression will be lossy. The degree of information loss will depend upon the heterogeneity of the agents grouped into the same containers. Attributes of the agents stored within a container are categorised in one of three ways. The attribute may be *compressible*, *storable* or *state-dependent compressible*. Compressible attributes are those that can be compressed (i.e. one copy controlled by the container for all agents). Storable attributes are those attributes of each individual agent that cannot be compressed. An example of an attribute that could not be compressed is a serial number as they are unique to each agent. State-dependent compressible variables are those variables that have certain values that can change frequently and other values that do not change frequently. Therefore the variable is only compressed when it contains a value flagged as not frequently changing. This is domain dependent and needs to be specified by the developer.

An implementation of this compression system in Java showed significant performance improvements when large numbers of agents were present in the simulation. In the sample model, up to 5% of the agents were heterogeneous and un-compressed. Agents transitioned frequently from compressed to un-compressed states resulting in an average turnover of 15 times during the course of the simulation. The results showed that when the simulation contained low numbers (100-500) of agents, the execution time was slightly higher than the benchmark that did not use the compression. This is due to the processing overhead of the compression manager. With larger numbers of agents, the execution times for the implementation using the compression system were significantly less. See figure 2.6.

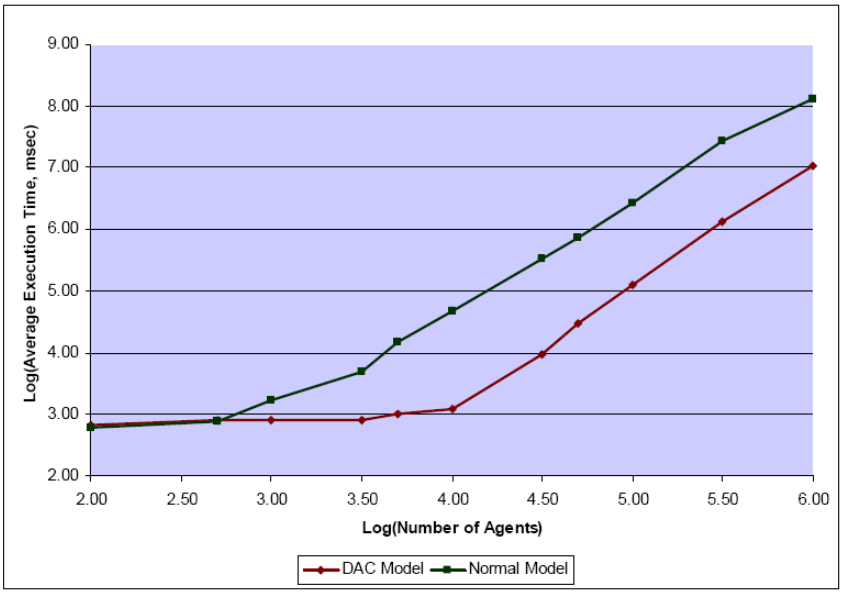

Figure 2.6 Execution Time of the Sample Model with and without Dynamic Agent Compression (DAC) (Source:[49])

2.5 GPU Technology and Its Use for General Processing

In recent times, Graphics Processing Units (GPUs) have shown themselves to be very powerful platforms for not only graphics-specific processing tasks but also general purpose processing tasks. A typical GPU can consist of hundreds of processor *cores* where each core can execute multiple threads. The graphics cards that the GPUs are mounted on provide larger amounts of very high band-width memory. This allows for larger scale parallel execution of complex pieces of software and offers an excellent platform for the development of complex agent based models. In this section the Nvidia Compute Unified Device Architecture (CUDA), ATI's implementation of the OpenCL software development environment and Microsoft's DirectCompute software development kit, all of which provide platforms for general purpose programming on graphics hardware, will be discussed. A series or agent based modelling platforms that are implemented using CUDA, OpenCL and basic GPU software developments kits will also be reviewed.

The Nvidia CUDA programming environment[10] allows software developers to write programs that use graphics card GPUs and memory for general purpose programming tasks, instead of just graphics processing tasks. GPUs are specifically designed for processing tasks that require a large proportion of arithmetic operations as compared to memory operations. Within the actual GPU, a large proportion of transistors are devoted to arithmetic processing which differs from general purpose CPUs that include cache and program flow control centres. The result of this design is that a GPU is suited to *data-parallel* programs where multiple instances of the same program are executed on multiple data elements in parallel. Figure 2.7 illustrates the difference in performance for floating point operations between general purpose CPUs and GPUs.

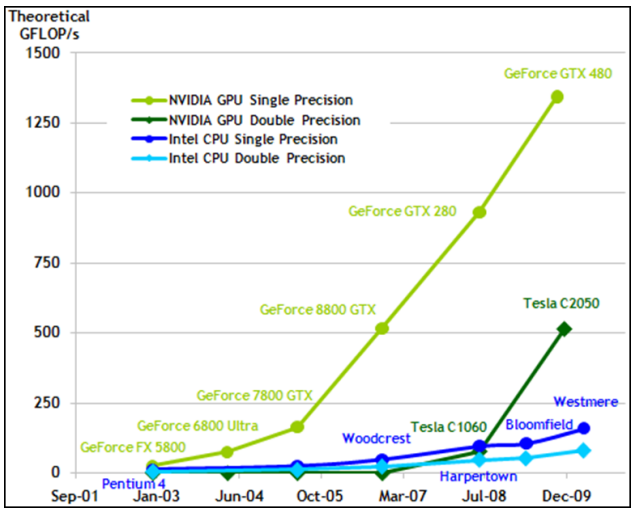

Figure 2.7 Floating-Point Operations per second for GPUs and CPUs. (Source:[10])

The CUDA programming environment extends the C programming language. It allows developers to produce parallel programs that can be executed on a range of Nvidia graphics cards. Each model of GPU has a different number of cores (some can have over 400 cores) and these cores are capable of supporting a number of threads. The programming environment hides the complexities of each different card and divides the threads into *blocks* of execution across the available processing units. Currently up to 1024 threads may be allocated to a single block and processing time is scheduled between these to compete the processing efficiently (see figure 2.8).

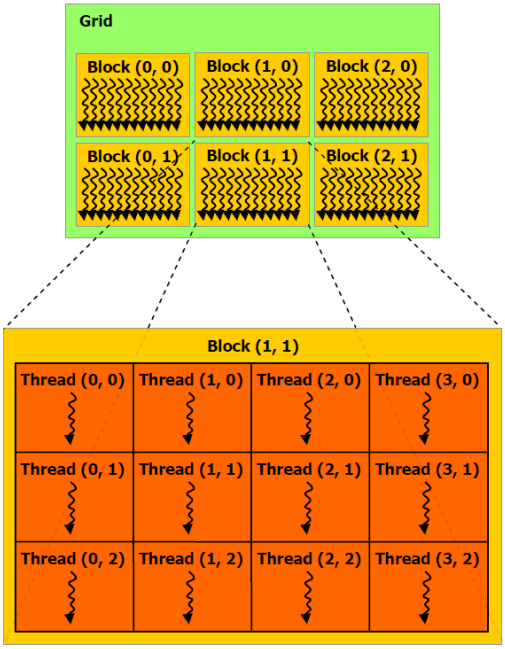

Figure 2.8 The logical structure of blocks and threads in the CUDA programming environment. (Source:[10])

The C programming interface allows a developer to define functions called *kernels*. These functions are executed N times in N threads simultaneously on a data item. This allows for massively parallel processing of large sets of data. The kernel functions must be designed in such a way that they work on their own part of

the data or overall problem. This means that effective partitioning methods must be employed to divide processing across the kernel functions when executing. This concept is similar to the parallel systems described in previous sections where a simulation's processing tasks are divided across different nodes in traditional parallel system.

In addition to the parallel processing capabilities of the GPU, graphics cards can also provide large quantities of high-bandwidth memory. The CUDA architecture organises the various sources of memory into a hierarchy that threads of execution can access. Each thread within the program has a private cache of local memory that can only be accessed within the thread. The next level up is a cache of memory accessible by all the threads in a block. The final level is a global memory space that all threads from all blocks can access. In addition to this hierarchy, *texture* memory and *constant* memory can be accessed and are persistent across thread executions. (i.e. if data is stored there in one execution of a thread, it will be available when if the thread is executed again).

A typical program written for the CUDA environment consists of sections of codes executed sequentially on a machine's main CPU and areas where massive parallel operations are carried out using the kernels executed on the GPU device. This is referred to as heterogeneous programming and allows a developer to construct applications that the best use of both programming paradigms. One of the key considerations when designing heterogeneous programs is the exchange of data between the host machine and the GPU device. Memory located on the host is not accessible to the kernel processes executing on the GPU device, and therefore before the kernels are executed any data from the host required for their operations must be explicitly copied across. This carries a level of overhead that needs to be factored in with program design.

Array Technologies Incorporated (ATI) has developed an implementation based upon the OpenCL architecture to operate on their GPUs that make use of their proprietary Stream Computing System[11]. The implementation follows a similar data-parallel architecture like that used in CUDA, however there are some differences between the systems that are derived from the differences in the underlying GPU hardware. The ATI GPU (referred to as a compute device) consists of multiple *compute units*. Each of these computer units contains multiple *stream cores* and each stream core contains *processing elements.* Figure 2.9 outlines this structure.

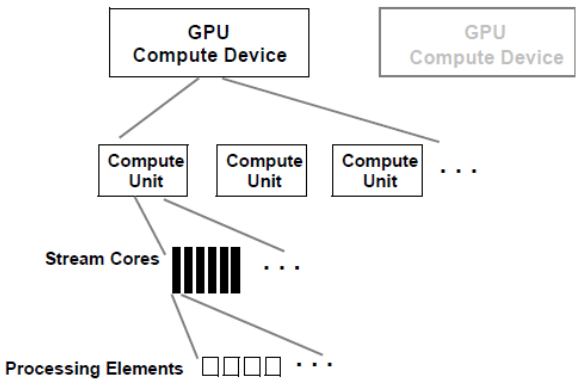

Figure 2.9 GPU Compute device structure. (Source: [11])

The stream cores are responsible for executing kernels, similar to those in CUDA, and the processing elements perform the basic operations such as integer and floating point calculations. The same kernel is executed on all stream cores from a given compute unit. Stream cores from different compute units can execute different kernels. An instance of a kernel running on a compute unit is referred to as a work item. The OpenCL architecture specifies how the work items are to be executed. Figure 2.10 outlines this process. The work items are organised into *work-groups*. The developer can specify how this is done on the available work items and this is enforced by the software. The work items from a work-group are executed in *wave-fronts*. Work items within a single wave front are executed in parallel and the total number of work items that can be executed as part of a single wave front is specific to the compute device that the programming is executing on. The stream cores (which are responsible for executing the work items within the wave front) execute the same instruction at the same time. In order to efficiently make use of the stream cores, multiple items from the same wave front are pipelined on each stream processor. This is primarily done so that the stream processors are not idle while work items are carrying out memory operations. Most GPUs allow up to 4 work items from the same wave-front to be pipelined on a single stream processor. This means that a compute unit of 4 stream processors can provide execution for wave-fronts with widths of up to 16 work-items.

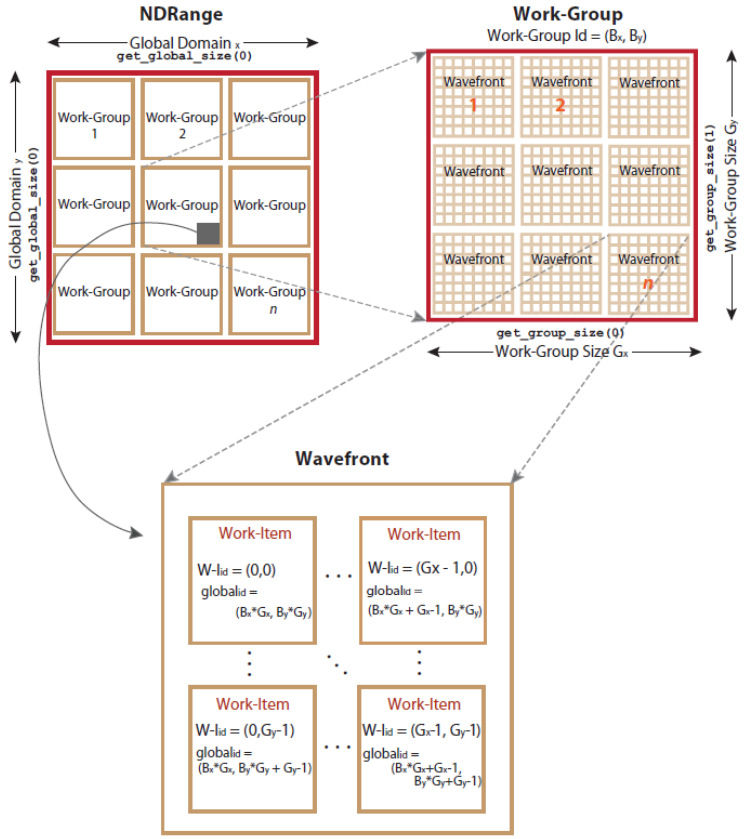

Figure 2.10 GPU Work item scheduling on the compute units. (Source: [11])

2.6 Agent -based modelling using General Purpose GPU Libraries

Before the introduction of Nvidia's CUDA and ATI's OpenCL implementations, GPUs were still used for general purpose processing tasks. This was done by making use of the functions provided by the OpenGL and Direct3D software development kits. The functions that these kits provide are intended for creating applications that contain 3D video output, such as computer games. The use of these functions for other purposes is referred to as *General Purpose Graphics Processing Unit* (GPGPU) programming [50].The work in [50] outlines a system that allows the OpenGL library to be used for data-parallel general purpose computation, similar to the Nvidia's CUDA and ATI's OpenCL architectures outlined previously. Using the OpenGL library can be more flexible than the CUDA and OpenCL systems because there is no standardisation amongst graphics card manufactures with regards to the general computing toolkits for graphics cards. This basically means that programs written for the CUDA system can only run on Nvidia cards and programs written with the OpenCL implementation can only run on ATI cards. GPGPU programs written using OpenGL can run on any graphics card that supports OpenGL, which includes practically every card on the market.

In the GPGPU approach in [50], *texture* memory on the graphics card, normally used to store the colour information for the 3D images to be displayed on the screen, is used as the primary memory by the graphics card for its processing. Data is copied from the computer's main memory, via OpenGL functions, to the graphics cards memory and back after it has been processed. The actual processing is carried out by *fragment shader* functions, which are normally used to calculate the texture colours for pixels of the frames to be displayed to the screen. In this situation, the shader function contains the general purpose logic of the program to be carried out. The shaders are used in the same way as the kernel functions in the CUDA and OpenCL systems; multiple shader functions are instantiated and they individually process a section of the input data independent of each other. OpenGL does not provide the same control over the execution of different threads on different cores of the GPU compared to CUDA; this is automatically done in the background.

The work in [51] outlines a framework for creating models that are capable for simulating massive number of agents. The framework uses the OpenGL graphics libraries to implement the framework and experimental results carried out as part of the research indicate a significant performance increase which is in line with the results observed in [9]. The work breaks the agents present in the models into two groups.

The first group is the *environment* agents, which have a static position in the simulated world and contain set of values that represent ambient quantities of the modelled world. This view of the environment as a set of agents is based on the ideas outlined in the Swarm simulation system outlined in [29]. Examples of these quantities include weather conditions such as temperature and humidity, or chemical concentrations. These environmental agents are represented by textures on the GPU. The quantities that the agent possesses are stored in the red, green, blue and alpha channels that would normally store the colour information for the texture. In this system, the information that would normally represent pixels in a frame to be rendered on a

video device is actually used to represent agents in the ABM. This texture information is packaged into a *frame buffer object*. Frame buffer object manage the streams of execution on the GPU. Like most agents in an ABM, these agents may need to update their state. In order to achieve this, a technique outlined in [50] referred to as *ping-ponging* is used. This technique involves using a fragment shader. A fragment shader is normally responsible for calculating the colour value of a pixel to be rendered, however, in this framework it is used as the update function for the agent. The fragment shader iteratively reads the information from the first frame buffer object, processes and passes it to a second frame buffer object. In the next iteration, the information is read from the second frame buffer object, processed and passed back to the first object. This occurs for every time interval in the simulation to update each environment agent. This process is parallelised because each thread in each GPU core can carry out this process simultaneously.

The second class of agents defined within the framework are the *mobile* agents. The mobile agents represent the elements in the model that move, interact and change state within the world represented by the environment agents. Due to the movement capability of these agents, they are significantly more complex to simulate on the GPU platform. The work in [51] specifies three basic operations that need to be carried out within the framework to effectively model the mobile agents. They include:

- Store the mobile agent in memory
- Update the state variables within the agent
- Connect the mobile agents to the environment agents.

Curiously, there is not specific mentioning of interactions between different mobile agents, however this may be encompassed within the update state task.

Mobile agents are stored using the same technique as the environment agents using the textures. The agent updating is implemented using the same ping-ponging technique outlined before for the environment agents, however each individual's behaviour must be handled by a different stream. Complex behaviour may also require multiple iterations of the ping-ponging technique within a single time step. For the third task (connecting the mobile agents to the environment) there are two methods described, depending on the direction of the interaction. The method used for the mobile agent to access the environment agent simply involves the mobile agent maintaining coordinate information for its current location. It can then access the environment agent through a double lookup process using the coordinates. It is more complicated for the environment agent to access the mobile agents. The work in [51] outlines a *scatter* operation. The details of this operation will be left out here but are available in [51].

The other key issues with implementing ABMs within the GPU parallel execution paradigm are execution ordering and avoiding collisions due to parallel execution. Execution ordering is handled by breaking the time step into series of phases. The example given divides the time step into an environment update phase, a movement phase and a replication phase. Each of these phases can be executed simultaneously across all agents in an arbitrary order within the time step. Handling collisions involves using priority system.

Essentially each agent has an associated with a priority value and if two agents attempt to carry out actions that are mutually exclusive, the action belonging to the agent with the higher priority is carried out. This is described as a *two-pass* system. This process may be carried out several times in a time step for each agent because an agent may still be able to carry out a valid operation even if its initial operation is pre-empted (i.e. the agent gets to try again to take another action if its first is blocked).

The work covered by Richmond and Romano in [52] outlines a library of functions, called the Agent- Based GPU modelling framework (ABGPU), for constructing agent-based models that operate on GPUs. The framework is built upon the OpenGL graphics library, allowing maximum flexibility and compatibility across a range of hardware. The aim of the system is to remove the need for a model developer to program the low level operations that map the agent to the stream programming model that GPUs use. The framework achieves this by allowing developers to define the agent's actions in a simple scripting language. The agents defined in these scripts are mapped to hardware operations on the GPU. The framework also includes functions of creating visualisations of the agents on the GPU hardware as well. In traditional CPU based modelling systems, the GPU is only used to render visualisations and not for model calculations; these are left for the CPU to carry out. This architecture carries the significant overhead data needing to be copied from the machines main memory to the graphics memory. By carrying out both the modelling and visualisation on the GPU, the data copying bottle necks are removed and the full potential of the GPUs data-parallel processing capabilities can be utilised for the modelling.

The system uses a similar model to that presented by Lysenko and D'Souza in [51], where the developer defined agents (through the use of agents scripting) are stored in the texture memory, with their various attributes stored in the red, green, blue and alpha channels for each pixel. The fragment shader functions are then used to carry out the processing for each stored agent simultaneously during each step of the simulation. In order to allow for communication between agents, a partitioning scheme is used where the agents are divided up based upon the communication radius of the agents. The partitions are maintained using sets of pointers that allow agents to know the physical location in the hardware of the GPU along with their locations within the simulated world.

The system was evaluated with a modified version of the *Boids model* which was developed based upon an implementation by Reynolds in [53] to simulate the movement of schools of fish. The ABGPU implementation consists of the agents modelling component as well as standard GPU coding that implements a live visualisation of the model. Each agent contains 7 variables for simulating its behaviour according to the model as well as position components that represent the agent's location in the 3D world. A *goal* point in the 3D world is updated at each time step in the simulation. This goal point represents the location that the school of fish is moving toward. It changes based upon a collection of thresholds and random factors. The ABGPU scripting for each agent in the simulation implements the physics of the environment and the set of rules that govern how the agents move in regards to other agents (i.e. the behaviour of the school). The ABGPU implementation (see figure 2.11) showed a significant improvement in performance over that which Reynolds achieved in [53], rendering 16384 agents (fish) with a detail level of 1500 polygons at over 30fps.

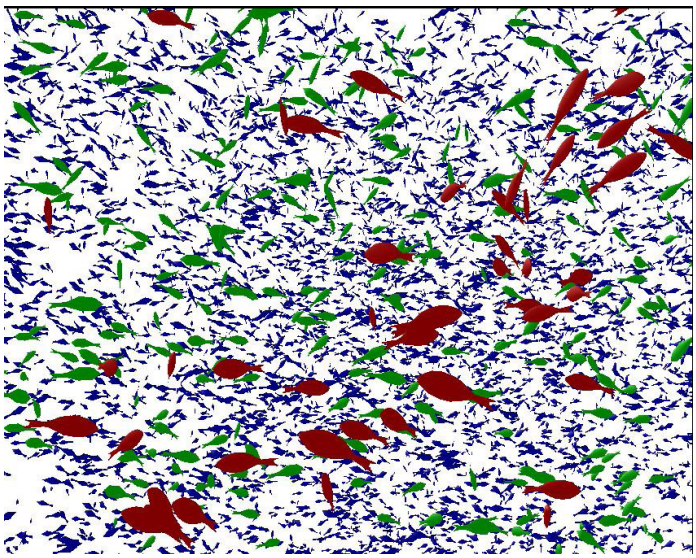

Figure 2.11 A population of 16384 agents with rendered with the LOD system. Red, Green and Blue colours represent three detail levels 0, 1 and 2 respectively. (Source: [53])

2.7 The Potential That GPGPU Technology Could Offer Agent Based Modelling

The ABM system outlined in [51] illustrates how the un-intuitive graphics libraries, which are intended for developing 3D applications, can be used to process the logic contained within agent based simulations. The benefits of using this approach are evident in the work by Perumalla and Aaby in [9]. The sample GPU implementations they develop show the possible performance improvements that can be achieved by using the GPU technology. Their work also exposes some of the limitations that the GPUs posses with regards to processing agent based model logic. In [9], implementations of three sample simulations are developed. They include *a mood diffusion model*, which models the diffusion of a mood amongst interacting people, a *game-of-life* cell model, which models a grid of cells that come to life and die according to the state of their neighbours and a *segregation model* where agents in a 2D grid adjust their position (based upon their neighbours) until they reach a 'happy' state. Two implementations for each model were produced; a standard CPU implementation using the RePast [32] and NetLogo [31] agent-based modelling toolkits and a GPU implementation of each model using the OpenGL graphics interface (not the new CUDA framework). In this implementation, each agent was mapped to one GPU pixel value and the state information is stored in the red, green, blue and alpha channels for the pixel, much in that same way as the agent information was stored in models developed in [51]. The software for the GPU implementation is a combination of Microsoft Visual Studio .Net and Nivida Cg toolkit code. The .Net code is responsible for running the sequential sections of the model and the Cg functions are responsible for the parallel sections of the agent code. This design reflects the heterogeneous programming style used with the CUDA framework described above but is achieved, however, with the use of the two completely separate programming environments.

Performance testing carried out using the three models was done on a machine with an Intel Core 2 Duo processor, 4 GB of memory and an Nvidia 8800GT Graphics card. The performance carried out measured the raw runtime speedup of the GPU implementation and analysed the effects due to conditional statements

and locality. Each test compared the GPU implementation with the CPU implementation for each specific model with increasing numbers of agents.

The results in [9] reports that the GPU implementations of the Mood Diffusion model and the Game of Life produced vast reductions in the runtimes, compared to the CPU implementations, as the number of agents increases. The Mood Diffusion model produced a peak of 30-fold reduction in runtime when 4.2 million agents were produced. It is speculated that the performance drop with a larger number of agents is most likely due to hardware artefact, such as memory swapping. The test involving the Game of Life model shows similar results, with a runtime reduction that scales well right up to the maximum number of 4.2 million agents, which gave a 16-fold speed up in comparison with the CPU implementation. The difference in the performance between these two models is explained by the conditional statements in the logic processed within each agent. As mentioned in the previous section, GPUs are optimised for carrying out arithmetic operations, by devoting a larger proportion of transistors to arithmetic processing and having a relatively smaller cache and a lack of systems such as conditional branch prediction. Due to this architecture, when a conditional statement is encountered in GPU executed code, each processor must incur the combined cost of the true and false branches, regardless of the branch actually taken. Therefore the Mood-Diffusion model produced higher runtimes compared to the Game of Life model, for the same numbers of agents, because each agent's logic contains more conditional statements.

The work in [9] also examines how locality of agents affects the performance by examining the runtime reduction of the of the segregation model, where agents exist within a 2D grid and can move to other surrounding (i.e. local) squares to achieve their happy state. In this model v is the number of squares that can be 'seen' by an agent in any direction, resulting in a maximum of $(2v+1)^2-1$ squares that an agent will need to examine in order to determine where to move next. GPUs perform well in situations where the application exhibits a high level of locality due to optimisations for processing 2-dimensional data than arrangements where processing occurs across localised sections. However as locality decreases, the GPUs lack of L1and L2 cache impairs runtime performance (compared to the CPU) and the decrease in runtime is far less significant. The results for the Segregation model reflect this trend. When v is increased, hence decreasing the locality within each agent's processing, the speedup provided by the parallel execution of the GPU falls significantly. With low values for v a speedup of around 70× is observed. When v is increased to 16, the speedup drops to just 16×.

2.8 Agent-Based Modelling Using Nvidia CUDA

In the previous section, the use of General Purpose GPU programming for agent based modelling was examined. The systems reviewed outlined novel ways to make use of the un-intuitive graphics library functions for agent based simulation, producing promising results. In this section, various agent based simulation systems, based upon different underlying algorithms are reviewed. The focus of this section is how the various simulation systems make use of the Nvidia CUDA GPU programming system which was outline in section 2.5.

The major challenges that has confronted researchers attempting to make use of the CUDA framework for agent-based modelling is mapping the logical structure of an agent based model, that is a series of entities interacting over a series of time intervals, into the data-parallel programming paradigm that is provided by CUDA. The work carried out by D'Souza et al. in [7] outlines a framework for agent-based modelling within the CUDA environment that has been applied to modelling the disease Tuberculosis. The agent based model is represented in the CUDA environment by a set of data that contains the individual agents' information and set of kernel functions that update this data during each time step within the simulation. The tuberculosis simulation that is implemented in this framework defines three classes of agents:

- *Environment Agents* are static agents that form a grid that the other agents in the simulations can move around. They store *bacteria* and *chemokine* levels which determine the behaviour of the other agents and update these values in every time step.
- *Macrophage Agents* are mobile agents that are represented as finite state machines (FSM) that exist within the environment represented by the Environment Agents. They progress through a set of different states that are determined by interactions with the environment and T-Cell agents. Their movement within the simulated environment is determined by the chemokine level in the Environment agents.
- *T-Cell agents* are similar to the Macrophage agents as they are mobile and represented by a FSM, however they only have two states.

The details of the algorithms used to update the states of these agents will not be covered here as this is not the focus of this thesis. The agent's state is updated during every time interval by a set of kernel functions. This design follows the data-parallel paradigm of CUDA by allowing a separate instance of each update function to run in its own CUDA thread. This means that each agent's state update runs completely in parallel with all other agents of its type. The challenge that this design introduces lies with allowing interactions between the different agents. In the tuberculosis simulation there are several interactions between agents, such as those between the macrophage and T-cell agents and the environment (e.g. to determine movement direction and speed). The framework allows for this by making use of global memory, synchronisation and conflict resolution. Reading information from other agents is relatively simple because CUDA allows data declared at different scopes to be read. Data that is required by different agents (hence in different update kernels) can be declared at the global level and accessed by any other kernel. By using the discrete ordering of different agent updates within a single simulation step, many possible race conditions that could occur from concurrent updating and reading are avoided. Conflict resolution is used in the kernel that is responsible for updating agent movement. This is achieved by buffering agent movements after they have been carried out in the parallel agent functions and then using a separate kernel to resolve the situation where multiple agents attempt to move to the same location. After the resolution is complete, the movements are actually updated.

The structure of the simulation follows the heterogeneous program design discussed in section 2.5 from the CUDA programming guide [10]. Basically the CPU run code initialises the simulation and manages the

actions performed by the GPU in each time step. Once agent data (from the previous time step) has been copied to the graphics card, the update kernels for the various agents are executed in a pre-defined order (show in figure 2.12) on the GPU. Data can either be copied back to the systems main memory at the end of each time step, allowing the CPU to perform reporting functions, or multiple iterations of the simulation can be executed on the GPU before the data is returned to the main memory. The scheme chosen depends upon data collection and display requirements. For example in a visual demonstration, information would need to be coped back to main memory after each time step so that the CPU can update the GUI, but in a batch processing situation data would only need to be copied back at the end of the simulation.

The performance results for the simulation show that it scales well when the number of initial macrophage cells in the simulation and the environments size are increased when compared with a CPU implementation of the simulation. Both the GPU and CPU implementations scaled well when the number of macrophages was increased (keeping the environment size constant) however the GPU implementation produced faster execution times. When environment size was increased, significant increases in execution time were experienced by the CPU implementation, whereas the GPU implementation produced significantly lower execution times for environments larger than 64×64 cells and scaled well with only a minor increase in execution time as the environment's size was increased to 128×128 and 256×256. (See figure 2.13)

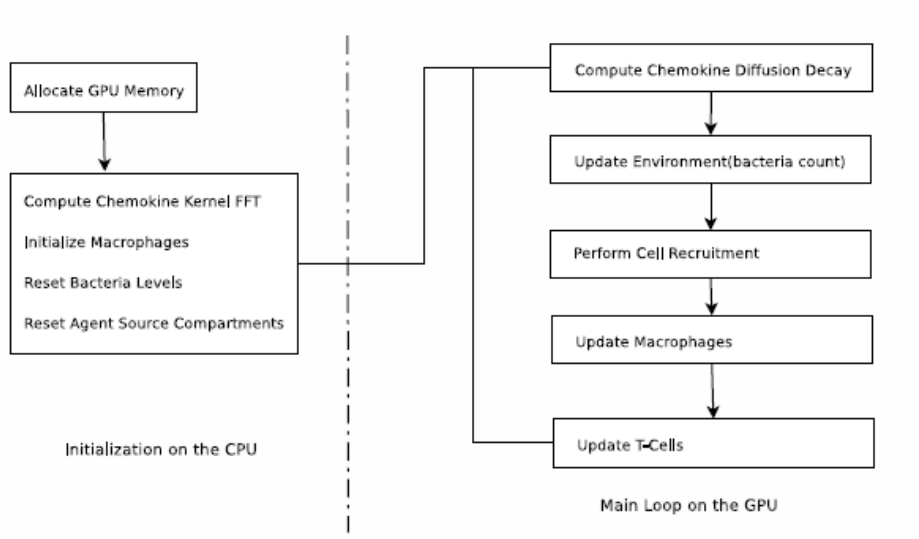

Figure 2.12 The data parallel simulation loop (source[7])

Strippgen and Nagel have developed a novel application of the GPU data-parallel paradigm to a queue-based, multi-agent traffic simulation in [5]. The simulation uses a queue algorithm, where agents, that represent the vehicles, move through a graph which represents the environment. The nodes in the graph represent major intersections and locations that the agents can travel to. The links represent the roads that the vehicles can move along. Each of the links in the graph is represented by a first-in-first-out queue that has a size, which corresponds to the number of vehicles that can occupy it, and a time value that represents the time it takes to travel along the link. A second queue for each is used to represent the number of vehicles that can leave the link in a given time period. Like the agents in the STREETS pedestrian simulation [1], each of vehicle agents in this simulation have a predefined route that they follow across the set of nodes and links.

This route takes into account where the agents 'home' is and 'work' activities along with other places in the environment that the agents may travel to.

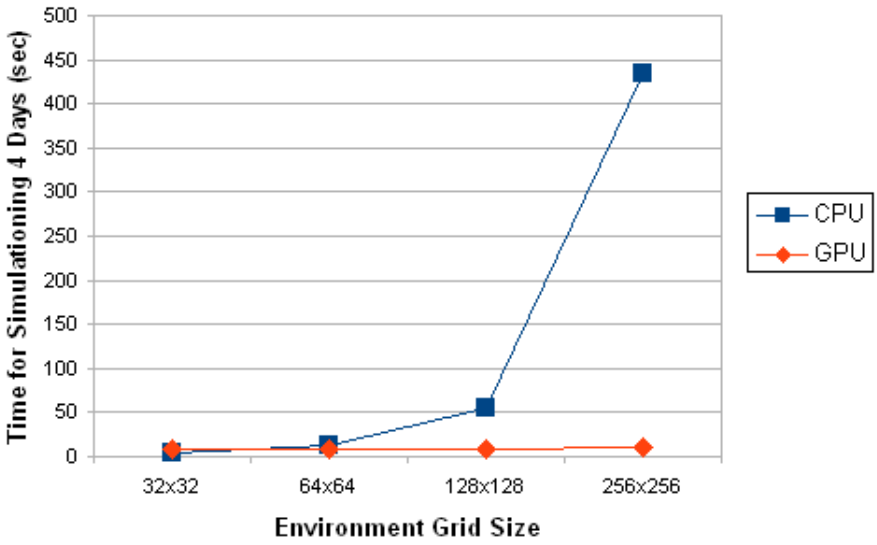

Figure 2.13 Performance Scaling with environment size. (Source [27])

The simulation uses the CUDA's kernel functions to update the state of the queues. In order to avoid the race conditions caused by simultaneous updates, there are two separate kernels for the movement of the agents through the graph. One set of kernels processes the agent's movement through the queues that represent the road links in the simulation. A second set of kernels process the agent's movement across the nodes from the output queue of the link connected to the node, to the input queue of the link on the other side of the queue. When an agent is at a location (for example 'work'), it is removed from the graph as it is not taking up space on any of the queues. A third set of kernel functions is used to process the agent's route through the graph by deciding when the agents need to be inserted and removed from the graph at each time step. This queue based system has been used in other traffic simulation systems such as the system covered by Cetin et al. in [54]. The simulation system developed in this work was designed to run on a cluster of computers connected by a high speed Myrinet network connection. Despite the similarity of the underlying algorithm used in both implementations, the CUDA system produces better performance figures for simulations using the same number of agents. Strippgen and Nagel speculated in [5] that this is due to the network communication overheads of the cluster implementation, such as the message passing required for agents communication and information replication on each node. The raw clock speed of most GPUs is also faster than that of most CPUs [10] and this also contributes to the improved performance of the GPU implementation.

The work outlined in [55] by Passos et al introduces a system for simulating crowd behaviour (for any species) of over a million individuals at interactive rates. The system provided a framework for simulating any form of *crowd* behaviour including flocks of animals and human pedestrians. In crowd simulations, an agent's behaviour is dependent on the locations and behaviour of other agents that are within a specified distance to the agent. This is known as the agent's *neighbourhood* and follows the concept as the neighbourhood in cellular automata systems. An example of this concept is the extended Moore

neighbourhood used in CA, presented in [34], where an agents/cell's neighbourhood consists of the surrounding agents or cells within a defined radius. See figure 2.14.

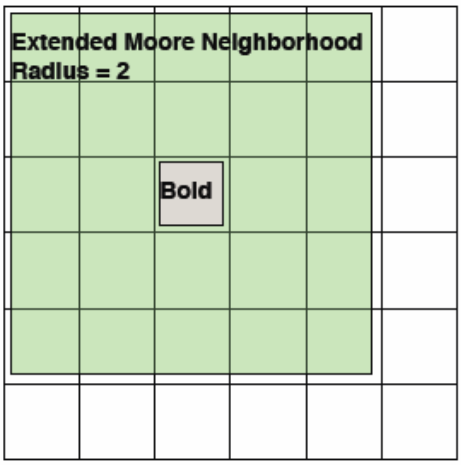

Figure 2.14 Extended Moore Neighbourhood. (Source [55])

The discovery of an agent's neighbours requires distance measures of all agent pairs, resulting in a $O(n^2)$ operation. In addition, this operations need to be performed at every time step due to the movement of agents within the environment resulting in a very computationally expensive task. The implementation presented in [55] overcomes this problem by using a sorting technique based upon the position information contained in each agent. Each agent in the simulation contains a basic set of information, including a 2-dimensional vector that represents its position, a 2-dimensional vector containing the agent's movement direction and velocity and an integer containing the agent's type information. This information is stored in a set of matrices, and the position matrix is used to sort the agents so that their neighbours (within a given radius) are arranged in the cells surrounding the agent. Figure 2.14 shows an example of this where the agent at position (2,2) will only have access to the 24 surrounding agents. This use of this sorting system eliminates the need for complex (i.e. $O(n^2)$) proximity queries as the agents that are within the specified radius are stored within the cells surrounding each agent.

The structure of the simulation is similar to that in the system presented in [7], where the CPU is only responsible for the initialisation and finalisation processing. The main simulation loop runs on the GPU with various kernel functions called during each cycle that carry out the actual simulation logic. The simulation cycle is divided into a *sorting pass*, where a sorting algorithm performs the positional sorting based upon the position matrix, and a *simulation pass* that is responsible for implementing the agent logic. The framework is capable of implementing any crowd behaviour algorithms within the simulation pass and simulation pass can be broken down into as many CUDA kernel functions as necessary.

The sorting algorithm sorts the agents, based upon the entries in the position matrix so that smaller values for the X and Y co-ordinates end up on the top leftmost corner and the agents with the highest values for X and Y end up in the bottom right-most cells. Any sorting algorithm that can achieve this can be used. The one implemented in [55] is similar to a bubble sort that is carried out in two dimensions on odd and even rows

and columns (resulting in a 4 step process). The algorithm starts by checking the X values for the entries in the even columns against their immediate neighbours in the odd columns. If necessary, the agents are swapped (i.e. the positional information) as well as the corresponding agent information the other matrices is swapped. This is carried out for the odd columns and then the odd and even rows in the position matrix. The result of this is a traversal of the whole data structure in $O(n)$ linear time. In addition to this, the odd and even steps can be executed in parallel, however this is not implemented.

This simulation framework produces promising results when using benchmark simulations. Tests were carried out on an Nvidia 8800GTS graphics cards, containing from 64,000 thorough to 1,048,576 agents. The benchmark contained 8 different kinds of agents. For experimental purposes a CPU implementation was also tested. The average time to compute a single cycle (i.e. frame) of the simulation was recorded from 10 runs with different number of agents. The graph shown in figure 2.15 illustrates the results.

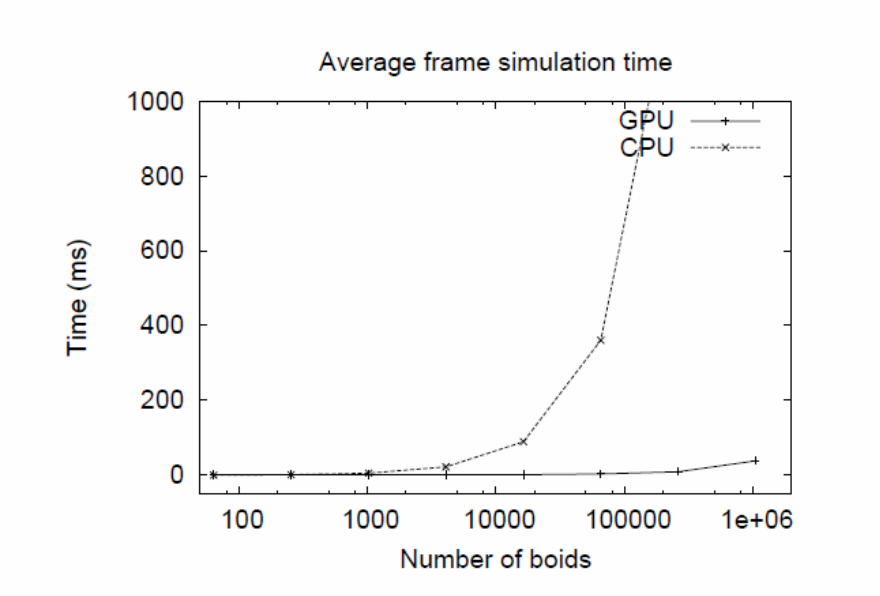

Figure 2.15 CPU vs GPU Results.(Source: [55])

Figure 2.16 shows that the number of agents in the simulation heavily affects the speed of the CPU implementation where the GPU implementation only shows a minor increase in the time for a simulation cycle.

So far in this section, the systems reviewed have followed the heterogeneous programming pattern (discussed in section 2.5) where the CPU carries out initialisation and finalisation steps and a single GPU is responsible for executing the simulation logic. The work by Aabyet et al. in [56] presents a system that allows execution of a model over multiple GPUs on multiple computer systems. The system implements a latency hiding, parallel system to offset the communication overhead between different computer nodes and different CUDA threads within a GPU. The system is designed to operate with cell-type agent based simulations, where the agents are represented in a grid and interact with their neighbours to simulate the phenomena, much in the same way as a cellular automaton. The matrix of agents is split up into blocks of $B \times B$ agents and each block is processed on a different processing element (either a Block or a CPU thread).

The problem with this division is the communication between the agents on the boundaries of each block. These agents will need to communicate with agents in other blocks for their state updates. This communication adds overhead to the simulation. In order to avoid this, the work in [56] uses a latency hiding system that allows the agents around the $B \times B$ block to be *padded* by R layers of data (see figure 2.16).

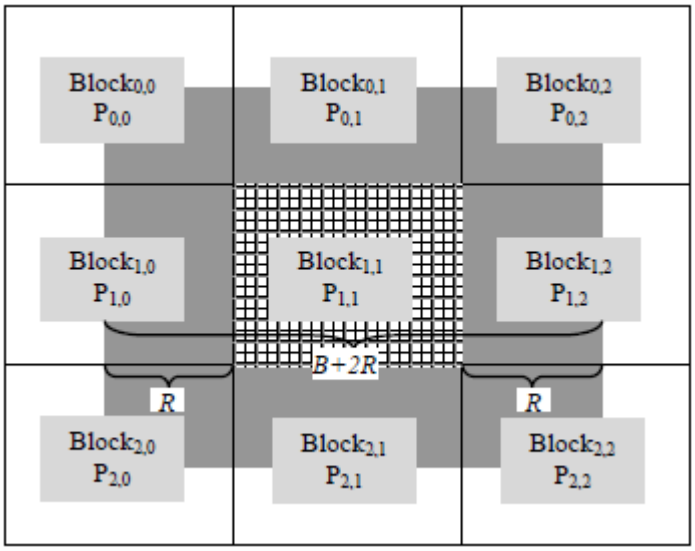

Figure 2.16 The B+2R Scheme for latency hiding.(Source: [56]).

These layers of data encapsulate the agents in neighbouring blocks, being processed on other processing units. The R layers of data allow the local agents to be increased R iterations before the padded, off-processor, agents data have to be re-synchronised with the actual agent data that has evolved after R iterations on the separate processing units. The benefit of this system is that it means that communication between the blocks of agents on different processing units only needs to occur every R cycles of the simulation, hiding the latency by allowing the processing based upon the un-synchronised data.

This scheme is implemented on a cluster of 64 bit Linux machines. One implementation was developed that used only the CPUs on the nodes and another used the GPUs installed in each node. The CPU execution uses POSIX threads for agent logic and the GPU implementation allocates a CUDA thread for the execution of each agent's code. The cluster of machines used MPI as the platform for inter-node communication for both implementations. The GPU implementation works by dividing the agents evenly amongst the nodes in the cluster and then dividing those agents into blocks based upon the maximum physical block size. The blocks must also allow enough memory for the padded agents for the processing. The agent logic is processed by CUDA kernels that read the agents block information which is stored in the global memory on the GPU. In order to improve performance, the agent block information is copied to the block level shared memory. The kernels execute their logic R times, then copy their $B \times B$ agent blocks back to the global memory for synchronisation. Each node's information must also be propagated to other nodes in the system for complete synchronisation of all agent data. This is carried out by POSIX threads on the nodes CPU that make use of the MPI message passing function. The CPU implementation uses the same basic scheme, except each agent does not have its own thread. Instead, each block of agents is further divided up and a single thread is

responsible for the execution of multiple agents' code.

2.9 Modelling Biological Invasions

This section introduces the concept of a biological invasion and reviews several methods for simulating biological invasions, including the population and spatial characteristics. This section includes a review of the CLIMEX modelling system and the existing bio-climatic modelling of the OWSWF.

2.9.1 General Characteristics of Biological Invasions

A biological invasion is the situation where any species of organism arrives or spreads to a location that it did not previously inhabit. There are literally hundreds of specific examples of biological invasions. This definition of a biological invasion includes the propagation of plants and animals into new eco-systems through migration and reproduction [57], the spread of human viruses such as *HIV* and *influenza* and animal viruses such as *myxoma* in rabbits and *rabies* in foxes. Mollison [58] explains that diseases can be viewed as predators of a larger species. Mollison also [58] provides an example where the rabies virus can be seen as an invader whose 'food' is foxes. Although there are a range of different types of biological invasions, they are all [28, 57-59] characterised by a similar set of stages that the invasion progresses through. By breaking the invasion down, different stages of the invasion can be modelled using different methods. This allows for an effective model that captures all aspects of an invasion to be developed.

Williamson, in [57], provides a conceptual framework for the study of biological invasions. The framework can be used to characterise biological invasions involving most types of organisms. The framework breaks invasion into 4 aspects:

- The *Arrival and Establishment* stage involves the initial introduction of the foreign organism to the new environment. This includes the mechanism that allows the organism to physically reach the new environment and the factors that determine if the invading organism can survive in the environment.

- The *Spread* stage involves the organism's ability to populate the new environment and spread throughout it. The spread can be viewed as a *front*, which represents the distance from the initial establishment point where the organism is present, or as an area (broken down into sub units) that has different population densities throughout. The spread of an organism can be determined by any number of factors ranging from the physical movement speed of the organism through to competition for resources.

- The *Equilibrium* stage follows the spread stage where the organism reaches a state where its geographical spread and population density are relatively static over time. In [59], Williamson outlines that population of some organisms can limit their own density, through internal competition, or through interactions with the environment and other organisms. If a species can maintain an equilibrium state for a significant time, factors such as genetic variance and evolution can affect the population.

- The final aspect of an invasion is the *Impact* on the environment. This includes areas such as the effect on the food chain within the environment, the effect on human activity, the costs of eradication and control measures and physical damage to the environment, such as soil damage etc. which are caused by organisms, into new populations that they can infect.

2.9.2 Predicting Population Growth with Equations

The simplest type of modelling used for the growth of populations in biological invasions involves using mathematical equations. These mathematical equations involve several variables and predict the growth of a population of the organism based upon their values. This type of model does not, strictly speaking, actually model an actual biological invasion as it does not take into account spatial aspects of an invasion and the biological factors that determine the establishment of the species in a new area. This type of model is usually referred to as a *simple-demographic* model and can be incorporated into full invasion models to calculate population growth within areas [28]. The most common growth equations used for biological modelling are the *Exponential* and *Logistic* models. The Exponential growth equation models population growth from reproduction. The calculation is based upon the population of reproducing individuals increasing in each generation (i.e. as new individuals are produced they begin reproducing in parallel with those of the previous generation). Because of the increase in reproducing individuals, the rate of growth is exponential according to the equation (2.1).

$$\frac{dN}{Dt} = rN \tag{2.1}$$

Where N is the population size at time t and r is the intrinsic of population growth, usually estimated from relevant data sources. This model for population growth does not take into account environmental factors that can limit or increase population growth, such as predation and migration of individuals. As a result it can only be applied to situations where the main, if not only, factor that determines growth is reproduction [60]. The exponential model assumes that the population's growth is infinite, which in reality is not the case. The logistic model addresses this issue by introducing a value, K, which is the population's *equilibrium* state. K is the population size where the growth is zero. The resulting growth equation (2.2) is shown below [60].

$$\frac{dN}{dt} = rN(1 - \frac{N}{K}) \tag{2.2}$$

Where N is the population size at time t and r is the intrinsic rate of population growth, usually estimated from relevant data sources. A modified version of this equation, referred to as the *logistic-difference* equation, deals with discrete numbers of individuals. The logistic equation still suffers from the other limitations of the exponential model and as a result its use is limited.

A more complex system of equations that predict population growth was devised by Leslie in [61]. The equations are represented by a set of matrix operations that can be used to describe population dynamics. The model calculates the number of female individuals from different age groups within a population at different time intervals based upon the *average reproduction rate* of the females in each age group, the *average*

survival rate of the females in each age group and the *initial age distribution* within the population. The model starts with the number of females in each discrete age group of the population. The partitioning of the population into age groups will depend upon the species in question. So at time $t=t_0$, the vector $\boldsymbol{X}^{(0)}$ is the initial age distribution vector and contains the number of females in each age group at t_0, where z is the number of age groups analysed.

$$\boldsymbol{X}^{(\mathbf{0})} = \begin{pmatrix} \boldsymbol{x}_{\mathbf{1}}^{(\mathbf{0})} \\ \boldsymbol{x}_{\mathbf{2}}^{(\mathbf{0})} \\ \vdots \\ \boldsymbol{x}_{\boldsymbol{z}}^{(\mathbf{0})} \end{pmatrix} \tag{2.3}$$

As time progresses the number of females in each age group changes according to the average survival rate, P_i, and average reproduction rate, F_i, within each age group. Therefore the number of individuals at time K from age group 1 can be calculated using the following

$$X_1^{(k)} = F_1 X_1^{(K-1)} + F_2 X_2^{(K-1)} + \cdots + F_Z X_Z^{(K-1)} \tag{2.4}$$

From the result, $X_1^{(k)}$, $X_2^{(k)}$ ... $X_z^{(k)}$ can be calculated using P_i as shown below:

$$\begin{gathered} x_2^k = P_1 x_1^{(k-1)} \\ \vdots \\ x_z^k = P_{(z-1)} x_{(z-1)}^{(k-1)} \end{gathered} \tag{2.5}$$

These equations are expressed as a system of matrices as shown below.

$$\begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_3^{(k)} \end{pmatrix} = \begin{pmatrix} F_1 & F_2 & \dots & F_Z \\ P_1 & 0 & \dots & 0 \\ 0 & P_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & P_Z \end{pmatrix} \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ \vdots \\ x_3^{(k-1)} \end{pmatrix} \tag{2.6}$$

This model provides a more detailed analysis of the population's growth by separating the age groups and taking the survival rate into account as well as the reproduction rate.

The demographic models covered so far assume that there is no random variance in the population model. The models assume that the populations grow homogeneously according to their input parameters without any other affects. When the population being modelled is large, the effect of random differences within the population's growth is negligible however when the number individuals being modelled is small, heterogeneous population growth characteristics are significant [58]. The easiest way to add stochasticity to a mathematical demographic model is to allow the growth, death or birth rates to vary according to a variable whose value is drawn from a distribution. A simple example is illustrated by Higgins and Richardson in [28] where the logistic-difference equation is modified so that the growth rate is multiplied by a variable drawn from a distribution.

2.9.3 Spatial Models

The population growth models covered in section 2.9.2 aim to predict the number of organisms over time for a given geographical area. They do not, however, attempt to model the spread of the organisms through this area. In order to completely model a biological invasion, the model must be able to simulate both the population's numeric growth as well as its geographic spread as the invading species populates new areas. Spatial models have been used to simulate phenomena in many different domains. Some notable examples of the application of spatial models include predicting land use patterns and urban sprawl [62, 63], the spread of disease in humans and animals [64, 65], the spread of agricultural pests [8], the growth and dynamics of forests [2, 28], pedestrian movements in cities [1] and the economics related to agriculture [6]. This part of the review will focus on spatial models that relate to the invasions of species (including diseases) as this is the focus of the work presented in this thesis. In a later section, the work relating to the economic impact of biological invasions is reviewed.

One of the simplest ways to model the geographical spread of an organism is through the use of spatial regression. The basic principle of regression is to use historical records to predict the future rate and extent of spread. An excellent example of this is the work by Perrins et al. in [66], which models the spread of several species of *Impatiens* plants throughout then British isles. Perrins et al. modelled the spread of the plant by dividing up area across the simulated area into *vice-counties* which had an average area of 2,211 $km^2$and used historical data, dating back as far as the 19th century, to determine the number of these that were infested at 20 year intervals. This information was then fitted to *Logistic* curves which describe the rate of spread by indicating the number of vice-counties that were infested at each 20-year interval. Figure 2.17 shows the plotted data fitted to the logistic curves for the number of vice-counties occupied over time.

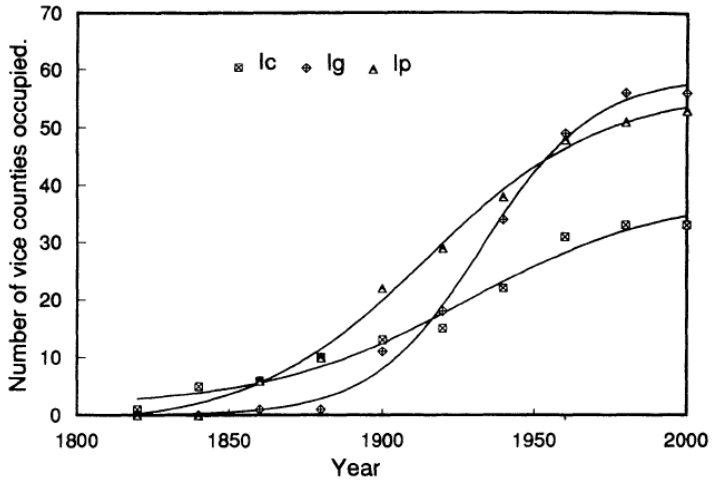

Figure 2.17 The number of vice-counties occupied the three different species of Impatiens over time. The points are the actual data and the lines are the fitted curves. (Source: [66])

Fitting the historical data to the curves simply involves adjusting the various parameters of the logistic

equation to produce a curve that mirrors the plotted points. Figure 2.18 below shows the logistic equation and the values for the parameters that were used to fit the curve to the data. It is important to note that the parameters that are adjusted to fit the curve to the observed values have no actual relationship to the biological processes that control the rate of the spread; they are simply parameters in the generic logistic equation that control the shape of the curve. This type of model is referred to as a Spatial-phenomenological simulation [28].

$$N = a + \frac{K}{(1 + \exp(-r(t-m)))}$$

where

N = the number of vice counties occupied
t = year
r = intrinsic rate of increase
K = asymptotic value
m = point of inflection on time axis
a = intercept on the axis N.

The parameters for the logistic equation for each of the three species are given below.

Species	Parameter	Estimate	s.e.	% variance accounted for
I. glandulifera	*r*	0.056	0.009	99.1
	m	1933	2.91	
	K	59	3.99	
	a	-0.06	2.25	
I. parviflora	*r*	0.035	0.008	99.1
	m	1915	5.83	
	K	58	8.25	
	a	-1.85	5.83	
I. capensis	*r*	0.031	0.010	96.9
	m	1928	10.4	
	K	37	7.33	
	a	1.62	2.84	

Figure 2.18 The Standard Logistic equation (for the curves in the previous figure) with the parameter values used to model the spread of each Impatiens species.

The spread of the plant is modelled spatially through radial expansion to the number of vice-counties specified at each 20-year time interval. In order for this method to work, there are a number of simplifications that are necessary. Each vice-county is considered to be square and of equal size which, in reality, is not the case. In addition, the first introduction of the plant to the vice-county is considered to be at the centroid point of the vice-county. These assumptions impact significantly on the model, however, due to the national scale of the model they are considered acceptable. A significant limitation on regression models of this type is the requirement of historical data. If enough data cannot be obtained to fit the regression curves, then it is not possible to generate an effective model.

Another class of spatial models, which have successfully been applied to simulate land-use changes, are first-order Markov models. Muller and Middleton [67] provide a good example of this type of spatial model with their simulation of land use change in the Niagara region of Canada. Markov models are similar to the

system of Leslie matrices reviewed in previous section as they use a system of matrices to represent the problem. The method used in [67] consists of three matrices which are iteratively multiplied to represent the time intervals. This is referred to as the Markov chain equation and is shown below.

$$\mathbf{M_{LC}} \quad * \quad \mathbf{M_t} \quad = \quad \mathbf{M_{t+1}} \tag{2.7}$$

$$\begin{bmatrix} LC_{uu} & LC_{ua} & LC_{uw} \\ LC_{au} & LC_{aa} & LC_{aw} \\ LC_{wu} & LC_{wa} & LC_{ww} \end{bmatrix} \begin{bmatrix} U_t \\ A_t \\ W_t \end{bmatrix} = \begin{bmatrix} U_{t+1} \\ A_{t+1} \\ W_{t+1} \end{bmatrix} \tag{2.8}$$

Where M_{LC} contains the land use changes for each possible transition for the time period, represented as a proportion based upon the analysis of topographic maps. (E.g. the matrix contains the proportions of land changes were from each state to another). M_t contains the land use (i.e. state) distribution at time *t* and M_{t+1} contains the land use (i.e. state) distribution at time *t+1*. This equation is applied iteratively to model the land use dynamics. The aim of this work is to represent land use dynamics; however this same mechanism of spatial state distribution representation can be used to represent the presence of plant and animal invaders. (i.e. the states can be the invaders present or not present). This modelling approach assumes that the transition to the next state only depends upon the current distribution and there are no historical effects. Like the regression models, there is no ecological or life-cycle based data used to model the changes in the system; only the empirical distribution data is used.

The final type of model reviewed in this section is the *Reaction-diffusion* model. Reaction Diffusion models make use of partial differential equations to model a population's expansion in a space as time progresses. Originally Reaction-diffusion models were developed to model the movement of substances in chemical reaction. Higgins and Richardson [28] provide an insight into the application of these models to the simulation of plant and animal spread. The basic form of the reaction-diffusion model that has been applied to model biological spread is summarised by the equation is shown below.

$$\frac{\partial u}{\partial t} = ru + D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2.9}$$

Where u(x, y, t) is the density of the organism at the spatial co-ordinates (*x,y*) at time *t*, *D* is the rate of movement of the individuals within in the population and *r* is the population growth rate. In this basic form, the model predicts the spread of an organism based upon the rate at which the population increases and the average movement rate of the individuals in the population. The actual spread that is modelled represents random diffusion into a homogeneous environment. This means that differentiation in the environment has no direct bearing on the rate of spread. This type of Reaction-diffusion model was first applied to plant and animal spread by Skellam in [68]. Skellam outlined a model for the spread of oak trees across Europe and Britain. By using empirical data from a number of sources, the values of the various input parameters were estimated and a model simulating the past spread was developed that could be used to make future predictions.

There are two main shortcomings, outlined in both [68] and [28], that the Reaction-diffusion model suffers from:

1. The model assumes that the dispersal of the organism is basically random. That is, there is no pattern to the direction of the simulated organism's dispersal. This random movement does provide a good approximation for the spread of a large number of organisms, where trends caused by the different behaviours of individuals and the environment are comparatively small within overall spread of the organism. (i.e. with large numbers the cumulative effects of a heterogeneous environment and heterogeneous spread patterns within the population are barely distinguishable from randomness).

2. The model's accuracy is dependent on the accuracy of the input parameters, in particular *D* and *r*. This can be said for any model, however because this type of model aggregates many factors into such a small number of parameters, this model is especially susceptible to inaccuracy of this type.

2.9.4 The CLIMEX Modelling System

The CLIMEX modelling system is a sophisticated, universal, simulation system that uses a range of climatic and ecological data to calculate the abundance of any given species in a location. The simulation model was originally developed by Suthurst and Maywald in [69], but the modelling system has been updated numerous times and is now up to its third complete version outlined in [21]. The model can be used to simulate the population growth of any species for which the required data can be obtained or estimated. CLIMEX uses a series of *indices* to calculate the potential growth or decline of a population for a specific location. Weekly climatic and ecological data is used to calculate the indices which are in turn used to calculate an annual *eco-climatic* index, which is a value between 1 and 100 that describes the climatic sustainability of location for a specific species. This index is designed to be an easy-to-use measure for policy makers and non-scientific users of the model. This is important as CLIMEX is commonly used as a decision support tool to estimate the possible distribution of a species to environments that they are not native. The CLIMEX system does not actually model the spread of the organism over time but rather the suitability for specific locations at a point in time, based upon yearly climatic averages.

The *eco-climatic* index is a function of the annual growth index, the annual stress index and the stress interaction index:

$$EI = TGI_A \times SI \times SX \tag{2.10}$$

Where, TGI_A is the annual Thermo-hydrological growth index, *SI* is the annual stress index and *SX* is the stress interaction index.

The annual growth index is calculated according to the function:

$$GI_A = 100 \sum_{i=1}^{52} \frac{GI_{W_i}}{52} \tag{2.11}$$

Where, GI_w is the weekly growth index calculated according to the function:

$$GI_w = TGI_w \times BI_w \tag{2.12}$$

Where, TGI_w is the weekly thermo-hydrological growth index and BI_w is the biotic index, which reflects the effects of a second species on the growth index.

TGI_W is calculated according to the function:

$$TGI_w = TI_w \times MI_w \times RI_w \times SV_w \times LI_w \times DI_w \tag{2.13}$$

Where TI_w is the weekly temperature index, MI_w is the weekly moisture index, RI_w is the weekly radiation index, SV_w is the weekly substrate index, LI_w is the weekly light index and DI_w is the weekly diapause index. The specific details for calculating these indices will be left out from here for brevity. These indices are calculated based directly on the climatic and ecological data put into the simulation. The stress index (SI_A) represents the factors that limit the growth of a population and the stress interaction index represents the combined effects of two stress indexes acting simultaneously. The full details of these indices are outlined in [21].

The CLIMEX model has been applied to many species around the world. Two basic examples include the prickly acacia in Australia, outlined in [70] and the soldier bug outlined in [71]. The work illustrated in these examples use the CLIMEX model to predict the distributions of their respective species in areas that they are not currently native in.

2.9.5 A Biological Simulation of a Screwworm fly invasion of Australia: An application of the CLIMEX Platform.

As part of a national preparedness strategy in the 1990's, the Queensland (Australia) Department of Primary Industries developed a national model for an invasion of Australia by the old-world screwworm fly (OWSWF). This work, outlined by Mayer et al. in [20], resulted in a sophisticated hybrid, spatial model that simulates the spread of the OWSWF across the Australian mainland. The model is discrete time cycle, cellular-automata type simulation with the Australian mainland broken into 20km x 20km squares that each possesses their own set of climatic and ecological data. The density of SWF present within each of these is calculated based on a weekly simulation cycle by a multi-module system. The model's hybrid structure consists of three main components:

- The Spatial Analysis System (SPANS), which is a geographical information system that is used to display the output spread maps to the user and carry out sections of the geo-spatial processing.
- A general programming component, consisting of a set of procedures written in FORTRAN77 that is responsible for calculating SWF densities across the grid squares in the simulation.
- The CLIMEX modelling system which uses the ecological and climatic information from each grid square to calculate growth indices (as outlined in the previous section). These growth indices are then used to calculate the density of adult female SWF present within each grid square. This approach as used instead of a more elaborate life-cycle model as it is more computationally efficient.

The model operates by calculating the density of female adult OWSWF in each grid square using a combination of output from the CLIMEX model for the grid square and a range of addition data including densities of wild life, feral animals and livestock densities and wounding rates for these population. This data allows the model to calculate the number of available sites that eggs can be laid, and the density of OWSWF within the grid square. The calculations for this are left out for brevity, however they are explained in [20]. The number of available sites for OWSWF to lay eggs also affects the dispersal distance of the flies across grid squares that surround an infested square. The dispersal mechanism uses a random direction, with the median dispersal distance determined according to the host availability in the source grid square. A 5 x 5 grid square window is placed over the source grid square and flies are distributed over the squares in the window according to proportions calculated from Monte-Carlo simulations carried out separate to the simulation. The dispersal simulation does not attempt to model the spread of the OWSWF and spot outbreaks caused by the movement of infested livestock from one location to another. The reason for this, stated in [20], is that patterns of livestock movement could not be simulated and accurate data on actual livestock movements, such as the National Livestock Identification System (NLIS) movement tracking data used in simulation systems like the National Model for Emerging Livestock Disease Threats [4], was not available at the time. The inclusion of this spread pathway has since been cited as a possible area for upgrade[72].

The model also simulates the use of the Sterile Insect Technique (SIT) for eradication of the pest. The Sterile insect technique is achieved through the use of detailed scientific measurements from experiments carried out in South-east Asia. The model simulates the 'buffering' technique used to drive the frontier of the invasion by 'releasing' sterile male insects across the frontier grid squares. This is modelled by modifying the OWSWF densities in the cells where the SIT is used.

The CLIMEX model was ultimately adopted as the key simulation tool for OWSWF establishment and decision support after it was validated against a lifecycle based model developed in Atzeni et al. [73]. This model, which is comprehensively reviewed in chapter three, simulates that complete lifecycle of the insect and is based firmly on biological observation and measurements, as opposed to the CLIMEX estimation. The lifecycle approach was not adopted for the final simulation because it was to computationally expensive for desktop computers at the time and the CLIMEX approach delivered acceptable approximation.

2.10 Concluding Remarks

This chapter has reviewed a collection of relevant, current literature covering the areas of agent-based modelling, Available ABM toolkits, ABM performance enhancements, the development of ABM using GPU programming and the use of ABM for biological simulations. From this review it is evident there has been limited work in the area of National-Scale biological agent-based modelling, with few of the available modelling tool-kits optimised for large scale phenomena, especially within the Australian context. Performance enhancement techniques such as data compression and GPU programming have been successfully applied to many simulations, however their capabilities have been limited and the work focuses on modelling smaller scaled systems. This trend is also true for stochastic modelling systems where, due to the computational demand, the implementation of stochastic techniques for modelling large scale phenomena has been very limited. The existing modelling work for simulating OWSWF spread within Australia has numerous short comings, including the low resolution and the use of a deterministic index system in place of a lifecycle algorithm based approach. These shortcomings reduce the decision support capabilities of the modelling system and impact its ability to capture all aspects of the biological phenomena.

The work in this thesis builds upon this literature by applying the agent-based modelling technique along with performance enhancement technologies, such as data compression and GPU programming, to develop a high-resolution, stochastic model to support decision support tasks.

Chapter 3

Improving the Efficiency of Agent-Based Models using Data Compression and Aggregation Techniques

3.1 Introduction

In this chapter, the application of data compression and aggregation techniques to agent-based modelling is explored with the aim of evaluating its potential for providing a high-performance simulation platform that can meet the requirements for implementing a national scale agent-based model of an OWSWF invasion. In order to confirm the effectiveness of the agent compression techniques before the agent-based OWSWF model is to be built, a decision was made to assess this by using an experimental *National Model for Emerging Livestock Disease Threats* (NMELDT), which was an earlier model we have been involved in constructing by collaborating with the Australian Department of Agriculture, Fisheries and Forestry (DAFF)[22]. This modelling system (which will be covered at greater depth later in this chapter) is modified to make use of the data compression techniques outlined within this chapter and compared directly against the un-modified version to assess the effectiveness of the approach in terms of memory savings and run-time efficiency. The aim of the work in this chapter is to develop an ABM compression platform that is capable of running a modelling system and assess the resulting performance. The work started in this chapter is carried over into chapter 4 where a series of more specific sample experiments will be included to obtain an understanding of the overall suitability of the approach with a focus on implementing the OWSWF model.

Data Compression is a technique commonly used to improve the efficiency of applications in other areas of Computer Science such as in video and audio software. Compression essentially involves finding similarities within the data and applying efficient representations of the data [46]. This can either be done in a static fashion before the simulation begins or dynamically as the simulation progresses. The work in [47] presents a system whereby multiple agents are represented by an aggregate agent. The algorithm works by treating each agent as a point in a multi-dimensional space. Each dimension in the agent's space represents one of the attributes that the agents possess. Examples of these could include age, location co-ordinates, disease life cycle parameters etc. Clustering algorithms are used to partition all the agents in the space into groups (i.e. clusters) of agents that possess similar attribute values. From here, the centroid of each cluster is calculated and the attribute values of this centroid are used to define a single agent that represents all agents in the cluster. In addition to this, the aggregate agent stores an *expansion factor* which indicates the number of agents that the aggregate agent represents. This is a one-off static process carried out at the start of the simulation. There is no alteration done to the clusters and the aggregate agents over the course of the simulation.

Wendel and Dibble [49] builds upon this idea presenting a system for dynamically compressing agents, allowing them to move in and out of their compressed state as their internal states change during the course of a simulation. The system works in either a *lossless* mode, where no information from each agent is lost during compression, or a *lossy* mode, which provides a higher level of compression but results in some information loss due to the compression process. The simulation system consists of two components: a set of agent containers that represent the clusters of agents in simulation and a compression manager, which acts as a 'wrapper' around the set of agents by coordinating model interactions with the agent containers. The agent

containers contain a count of the number of agents that they represent, just as the system outlined in [47]. At the start of the simulation, the compression manager groups agents that are similar into containers. The agents that are particularly unique are left un-grouped. The agent containers behave exactly like the agents that they contain. The compression manager manages the queries to the agents by directing the queries to the appropriate containers or unique agents. The containers respond to the queries as if they were individual agents. As the internal states of some agents change, causing them to differentiate themselves from their groups, the compression manager removes them from their respective containers and instantiates them as unique agents. If a unique agent's attributes change to match those of the agents present within an existing container, the agent is added to that container.

The method used to group the agents into the containers determines whether the compression is lossy or lossless. If the agents that are grouped into a container are *identical*, that is all of their attributes have identical values, then the compression will be lossless. For agents that are *similar,* that is their attributes values are similar, they are grouped into the same containers and the compression will be lossy. The degree of information loss will depend upon the heterogeneity of the agents that are grouped into the same containers. Attributes of the agents stored within a container are categorised in one of three ways. The attribute may be *compressible*, *storable* or *state-dependent compressible*. Compressible attributes are those that can be compressed (i.e. one copy controlled by the container for all agents). Storable attributes are those attributes of each individual agent that cannot be compressed. An example of an attribute that could not be compressed is a serial number as they are unique to each agent. State-dependent compressible variables are those variables that have certain values that can change frequently and other values that do not change frequently. Therefore, the variable is only compressed when it contains a value flagged as not frequently changing. This is domain dependent and needs to be specified by the developer.

An implementation of this compression system in Java showed significant performance improvements when large numbers of agents were present in the simulation. In the sample model, up to 5% of the agents were heterogeneous and un-compressed. Agents transitioned frequently from compressed to un-compressed states resulting in an average turnover of 15 times during the course of the simulation. The experimental results showed that when the simulation contained low numbers (i.e. 100-500) of agents, the execution time was slightly higher than the benchmark that did not use the compression. This is due to the processing overhead of the compression manager. With larger numbers of agents, the execution times for the implementation using the compression system were significantly less. A detailed analysis of the results in [49] show that the potential benefits of the scheme can be even greater with specific agent based models.

The remainder of this chapter outlines the basic principles and concerns surrounding agent data and develops a hierarchical data compression scheme for use with agent-based models, based upon the existing work for agent-based data compression approach. Specifically, work presented here extends the study by Wendel and Dibble[49] by implementing this hierarchical scheme that takes advantage of redundancy across the agent data-set. The work also shows how these data compression techniques can be applied to large scale models,

improving the efficiency of the memory usage and agent-updates. The final sections of the chapter examine a trial implementation of this approach, using the NMELDT and a discussion on the pros and cons of the application of this approach for modelling the OWSWF.

3.2 Implementation Framework

The architecture of the system uses a centralised approach where the core simulation modules update and interact with the agent set via an agent management interface. The interface decouples the agent data from the individual agents, as it is no longer encapsulated within each individual agent. For example, instead of an agent storing its own set of data, the agent manager stores the agent's data (along with the data of other agents) in a single, optimised structure that implements the compression system. The individual agent simply contains a pointer to the location of its data within the central manager. This allows for the system to take advantage of data redundancy across the entire set of agents and in addition to this, access to the data via the individual agents is still a simple operation. The interface can be implemented using parallel processing technologies (such as multithreading and distributed computing) depending upon the nature of the simulation and assets available to the simulation system. The use of this approach does mean that this agent manager can potentially become a bottle-neck within the simulation; update operations will require serialisation in order to maintain the integrity of the data storage scheme. The idea of this approach is that efficiency of the aggregation and compression of the agent data will offset any requirement for parallelisation when updating and creating agent data (read access will not require serialisation).

Figure 3.1 outlines the general architecture that the agent compression system uses. It is important to note that this is a general outline. Individual simulation systems can make modifications, depending upon the nature of the agents, their interactions and the way they connect to the other components of the simulation system (for example, GIS and database systems). Figure 3.1 also shows the flow of information (indicated by the arrows) between the components of an example modelling system. The *Simulation system* components are representative of a typical agent-based model where the simulation logic is implemented in a general purpose programming language and interfaces with database systems and possible GIS packages. The key component within this scheme is the *Central Agent Manager* which is responsible for managing the interactions with the agent set and the dynamic compression of the agents. The agent set consists of a set of custom objects that can, in turn, contain a set of sub-objects, forming a hierarchy of agent data.

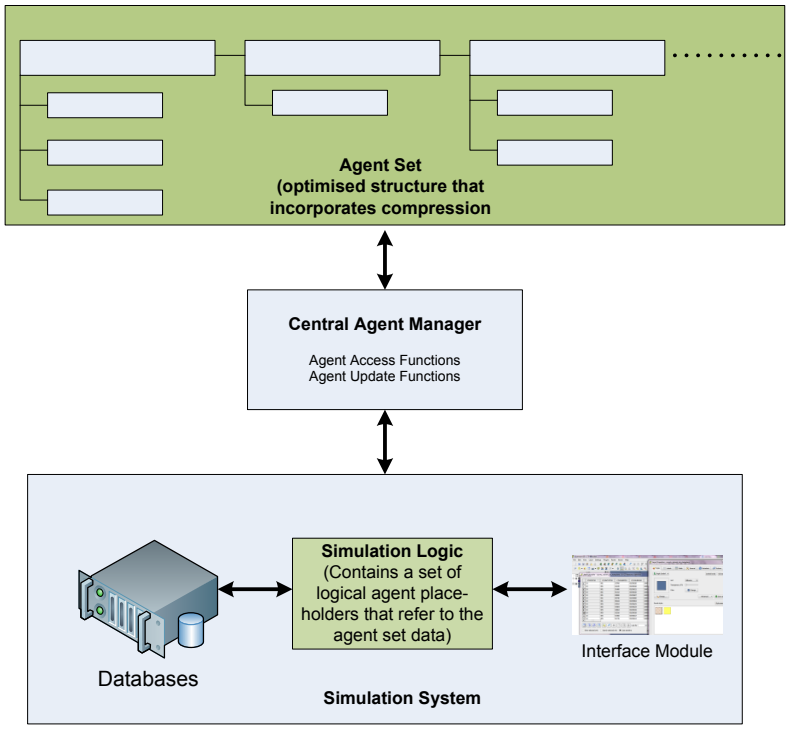

Figure 3.1 The high-level organisation of the components that make up the compression scheme.

3.3 Agents and Agent Data

The system proposed in this section is a form of loss-less compression that takes advantage of the inherent data-redundancy that is present amongst agents within a simulation. The agents in ABMs can be viewed as containers that hold state information. This information can be of any type imaginable and be of any size feasible, depending upon the requirements of the model and the phenomena being simulated. In the case study covered in the following section on NMELDT, the Disease Natural History (DNH) and physical state integers are examples of complex numeric data structures, however other examples common in ABMs include co-ordinates for positions, numbers of individuals and distances to neighbours etc. Strictly speaking, the state data stored within agents can be classified as *continuous* or *discrete* and within these categories it can be *dynamic* or *static*. Continuous state data can take any value within a range (for example, a temperature measurement), whereas discrete data can only take on a value from a finite set of values (for example, a count of individuals). Dynamic information changes over the course of time whereas static information stays constant as time progresses. These properties determine the level of redundancy and, in turn, how efficiently the data can be compressed. The diagram below (figure 3.2) illustrates the relationship across the level of redundancy, the nature of the data (static or dynamic) and the effect that loss-less compression will have on the data.

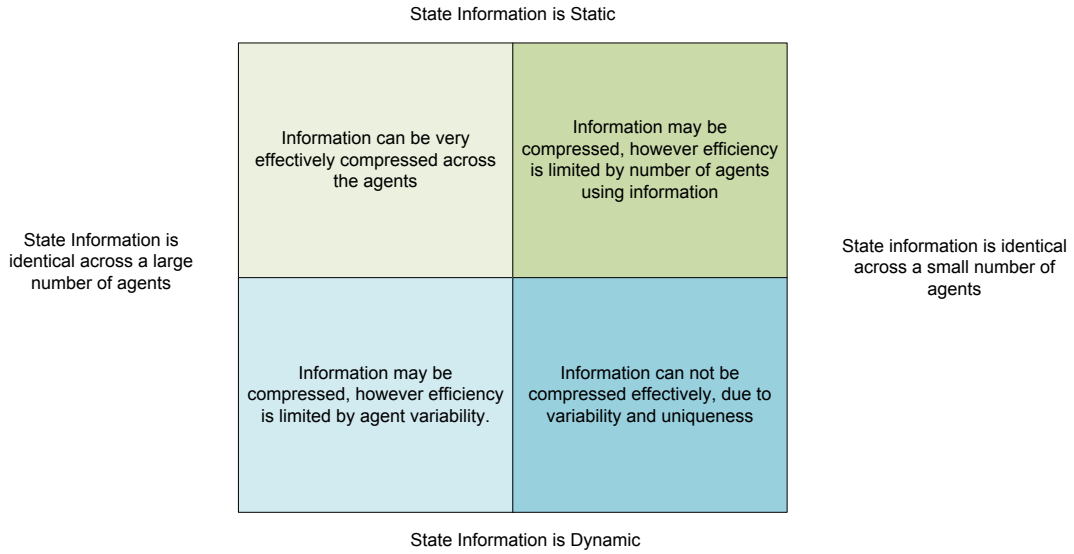

Figure 3.2 The efficiency of data compression techniques on data with different properties.

Data redundancy is a key concept when compressing data across multiple agents. In this context, redundant data refers to state data that is identical or similar (depending on the type of compression) across two or more agents. Compression schemes take advantage of redundancy by using an optimised representation of the redundant data within the structure or application that makes use of it. This principle can be readily applied across a set of agents within an agent-based simulation. At this point, it is probably prudent to point out that the agents within agent-based models are not necessarily homogeneous. Simulations frequently contain multiple types of agents, representing different aspects of the modelling domain. Each of these agents can have different sets of agent state data. The system for compressing state data in this chapter only deals with compressing a set of homogeneous agents, however the system can be implemented separately for each 'type' of agent in an ABM.

One key property of an agent's state data is its dynamic nature. In a typical ABM, each agent updates its state during each cycle of the simulation which, in turn, implies that the state data changes in each cycle. In addition to this, the agent's state data is often accessed by other agents and modules in the ABM. This poses a challenge to any compression system as the gains obtained from the compression of the data could very easily be out-weighted by the overhead of decompressing and re-compressing the data to allow it to be used.

3.4 A Hierarchical Data Compression Scheme

The first step to developing a compression scheme for an ABM is to divide and analyse the state data that the set of agents possess. This involves dividing the state data into separate items, determining which of these items are independent of others and ranking the data in terms of:

- Dynamic to static
- Redundant to unique

In this context, a state data is independent if it does not directly link to the agent's other data items and is updated according to separate functions. Based upon the properties of the data, a hierarchical data structure can be defined that contains the data based upon the combination of values present in each agent. The hierarchy is constructed by placing the least dynamic, most redundant data items at the top and moving down to the most dynamic, least redundant items at the bottom. The hierarchical structure has one level for each data item present within the agent type being compressed. By following the structure from the top layer to the bottom layer, the combination of values for each individual agent can be represented. Coupled with a *factor* value which stores the number of agents that the branch in the structure represents, this hierarchical structure can encapsulate the complete set of agents within the simulation. The representation and positioning of the agent data within the structure is completely independent of the events simulated within the model. Figure 3.3 displays the structure of the data within an example ABM based on this technique. In this figure, a simple hierarchy that represents a set of agents that have three data items is presented. These three items are:

1. *Movement distance*, which is a discrete numeric value in the range of 0 to 9 set at the time the agent is created.
2. *x coordinate*, which is an integer representing an x coordinate in a grid. This value is updated in each cycle according to an update function.
3. *y coordinate*, which is an integer representing a y coordinate in a grid. This value is updated in each cycle according to an update function.

Each logical agent in the simulation is represented by the combination of a top level data item (Movement Distance), one of its corresponding mid-level items (x coordinate) and the corresponding bottom level item (y coordinate), with the factor storing the number of items represented by the branch. This hierarchy has been developed by applying the mentioned technique; the movement distance is the most static, and due to its discrete nature and limited range, it is also the most redundant. The x and y coordinate values are placed below this in the hierarchy (their order is arbitrary as they are identical in nature) as they are more dynamic and have a greater range. This hierarchical scheme provides compression by taking advantage of redundancy amongst the agent set at each level (by each data item) in the scheme. For example, if two agents are identical then they will be represented by the same branch in the scheme and the factor variable is simply set to 2. The storage saving is made by the scheme because it only requires one set of information to represent the two agents. In this situation, two agents have been compressed into one representation. Similarly, if two agents only differ by one value, for example the y coordinate value, then the same *movement distance* and *x coordinate* data items will be used to represent that agent, with only the unique *y coordinate* values being required to represent the two. The scheme results in the maximum amount of compression for each state data item present in the agents.

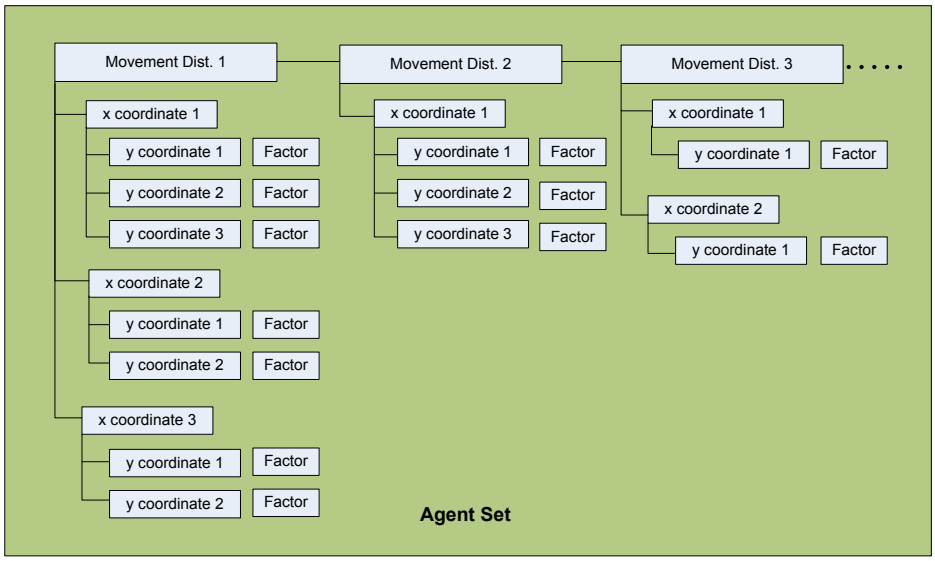

Figure 3.3 The Hierarchical data structure adopted to compress data in a sample agent set that contain three data items (Movement distance, x coordinate, y coordinate).

To show how compression is achieved, consider an example simulation with three agents of the type outlined above. The list below summarises the values stored in each agent:

1. Movement Distance = 5, x coordinate = 4.5, y coordinate = 6.7

2. Movement Distance = 5, x coordinate = 3.5, y coordinate = 6.8

3. Movement Distance = 5, x coordinate = 4.5, y coordinate = 6.7

Notice that agents 1 and 3 are identical and only the *x* and *y* coordinates differ in agent 2. Figure 3.4 illustrates the compression hierarchy that will store the set of agents in this state. This diagram shows how the hierarchy makes use of redundancy to reduce the amount of storage space required. Instead of storing 3 copies of the movement distance (all with value 5), one is stored at the top of the hierarchy. Furthermore, agents 1 and 3 are represented by the same sub-branch, requiring only one set of the *x* and *y* values to be stored. Assuming that the movement distance is represented by a 32-bit integer and the coordinates are represented by a 64-bit double precision type, without compression these three agents would require 60 bytes to store the state information. Using the compression hierarchy, only 36 bytes are required (the space required for the factor storage is left out of these calculations as it is negligible when large numbers of agents are stored.) The hierarchical structure is implemented using references to ordered sets of data items. The data sets are ordered as it allows for efficient access. The top layer consists of an ordered set of the top level data items. Each of these data items maintains a reference to the set of data items under it in the hierarchy and this continues down to the lowest level in the scheme. In addition to this, item sets maintain references to their parent items from bottom to top. In this way, the structure is completely linked up and down the hierarchy. The hierarchical agent storage system integrates into the simulation's logic through the use of place holders that operate within the simulation where, normally, the standard agents would have existed

with their complete set of data. These place holder agents simply contain references to the bottom level items within the compression hierarchy. This allows agent-wise access to the data stored within the compression scheme. Figure 3.5 outlines this integration, with the triangles representing the agents and the arrows representing the references to the agent data within the compression system.

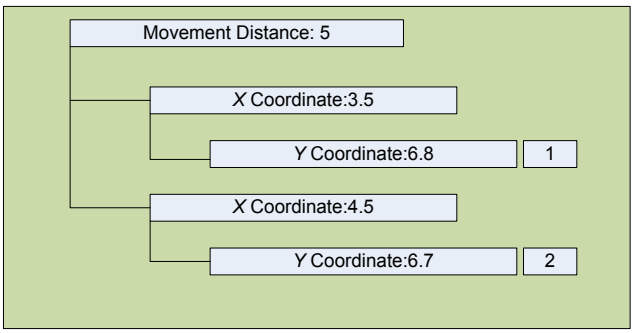

Figure 3.4 The compression hierarchy in a sample simulation run.

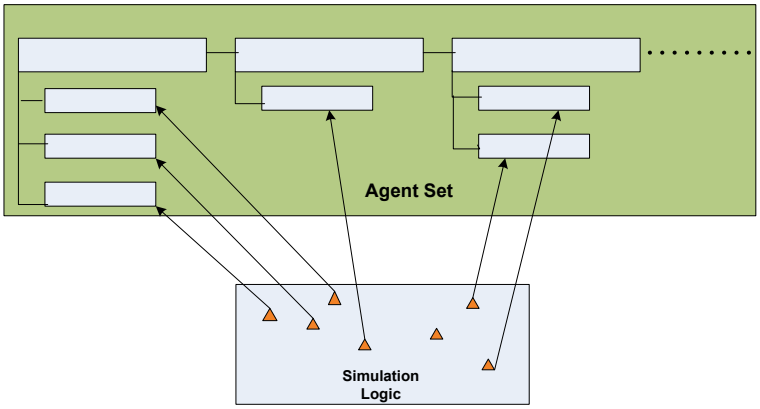

Figure 3.5 Place-holder agents within the simulation logic (represented by the red triangles). These place-holders maintain references to the lowest data item belonging to them in the agent data hierarchy. This allows agent-wise access to the agent data contained

3.5 Agent Access and Updates

The scheme that has been described provides an efficient way to represent the data contained within a set of agents, no matter how often agents change their state (and their state data) through the progression of a simulation. In addition to this, the agent state data needs to be accessible from other modules via the place-holder agents in the simulation. This means that the compression scheme must be able to provide efficient access to the agent data and the ability for agent data to be updated within each cycle of the simulation. These requirements are addressed by a combination of the operations provided by the *Central Agent Manager* and the references to the agent data that are maintained by the logical place holder agents in the system. As mentioned in the previous section, the individual agent place holders can access their data via the references to the bottom level items in the storage structure. By following the set of references that are maintained within the data structure, the complete set of data items for each individual agent can be accessed

by the place holder. This allows for agents to read and present their data to other modules and agents within the simulation. This deals with accessing data in an agent-wise fashion, but update operations are controlled by the *Central Agent Manager*. When an agent is updated, the values of its state information can change and the positioning within the compressed data structure must be altered. The exact details of the actual update operation performed on the individual agents will be different for each individual simulation and hence will not be covered in this chapter, but the algorithm for updating the compression structure is relatively consistent for all simulations. The basic steps for updating an agent's state and compressing this state within the hierarchical structure are captured in the following algorithm which is repeated for each agent in the system:

1. Retrieve the agent's state information from the compression structure, decrementing the factor value wherever necessary. Using the state update algorithm, update the agent's state information to the next state.
2. For each layer in the compression structure, starting with the top data items, search for a matching data item to the corresponding updated layer in the agent:
    a. If a match is found, move down to the next layer in the hierarchy or increment the factor value if this is the bottom level agent.
    b. If a match is not found on this layer, create a new branch in the hierarchy for this value, generating all branches below this layer and setting the factor value to 1 on the bottom layer item.
3. Repeat for the next agent.

This algorithm essentially involves decompressing the agent, updating its state, and then re-compressing the updated agent back into the hierarchy. The algorithm for decompressing agents data (used in step 1 of the access/update algorithm) involves reading the hierarchy from the bottom upward, following the references maintained within the structure. This procedure is summarised in the following algorithm:

1. Generate a blank agent structure to store the agents' information from the compression structure.
2. For each layer in the structure, starting with the bottom layer pointed to by the place holder's reference:
    a. Read the data item at the current layer to the correct location and copy the blank agent structure.
    b. If not the bottom layer, check for other branches from the current item.
        i. If there are no other branches, delete the item from the structure
    c. Update the current reference to the parent of the current reference.

This algorithm effectively removes the agent from the compressed structure and saves it in an uncompressed state.

3.6 Optimising the Agent Update Process

The update algorithm outlined in the previous section adds a significant amount of additional processing overhead, as each individual agent is de-compressed, updated and re-compressed. This process can be streamlined by updating agents in their compressed state, based upon the characteristics of the simulation. There are many situations where deterministic update operations that do not require cross-agent interactions are performed to the whole set of agents. For example, there may be a state update operation that determines the next state in a static manner based solely on the current state of each individual agent. This type of operation can be carried out on the complete set of agents without the need to de-compress each individual agent because each logical agent within the simulation will have an identical operation performed upon its state data. The process of updating the compressed data involves iterating through the hierarchy in a top-down fashion, updating data items by making use of the references up and down the structure. The exact nature of the update operation will be dependent on the data that needs to be updated. In the example developed in figure 3.4, the *x* and *y* coordinates are updated based upon the Movement Distance data. In this situation, the update function can simply update the values directly in the compression hierarchy (leaving the factor untouched) in order to update all agents in the simulation. This reduces the number of CPU clock cycles required when compared to the algorithm outlined in the previous section. Similar reduction can be achieved when compared to an identical simulation that does not use this scheme as each data item updated in the compression hierarchy is used by multiple agents. A more solid example of this optimisation will be covered in the following section in the case study of the NMELDT.

3.7 Case Study: Applying Data Compression to the National Model for Emerging Livestock Disease Threats (NMELDT)

The (Australian) National Model for Emerging Livestock Disease Threats, outlined in [4], is a large scale, agent-based model designed to simulate the spread of infectious diseases amongst cattle, sheep and pigs within Australia. The model has been developed in the Java programming language and integrates a suite of Java modules, a PostgreSQL database and an open source mapping package called OpenMap[74] to deliver a simulation system with a complete geographical information system (GIS) interface. The simulation software is designed to execute on a standard desktop PC; however the simulation data (provided by the PostgreSQL Database) can be hosted on a separate system and accessed via a network link. Figure 3.6 summarizes the high level architecture of the complete simulation system, with the arrows representing the flow of data between the major components. Figure 3.7, below, summarises the organisation of agents within the model. The basic agents within the simulation are *livestock* agents that represent individual sheep, cows or pigs in the model. The simulation can be required to model hundreds of thousands of these agents, so efficiency is essential for acceptable execution times. These agents are organised into groups that represent herds of

animals. The agents within each group are homogeneous, such that a herd can only consist of one type of animals. Premises including farms, saleyards and feedlots are represented by another class of agents that contain and manage the group agents. In Australia, every premise involved with the production of livestock has a *Property Identification Code* (PIC). The PIC is an eight character number identifying the location and ownership of a farming property.

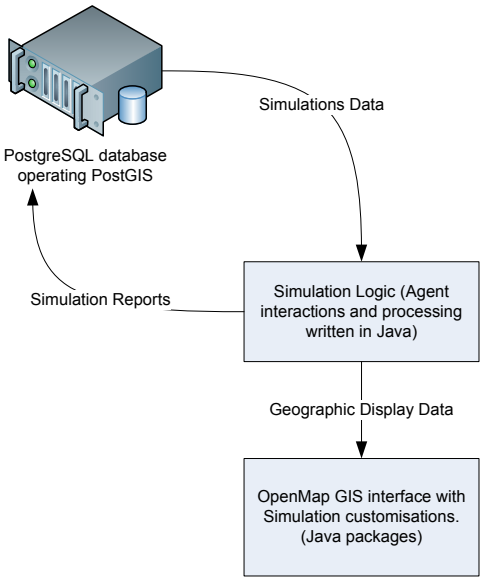

Figure 3.6 High-level overview of the National Emerging Livestock Disease Threats simulation system.

In order to accurately simulate the spread of diseases and pests that affect livestock spatially, the movement of the actual livestock must be simulated. This principle is based around the idea that if an animal, infected with a disease or infested with pests, is moved from one location to another, the disease or pests may be passed to uninfected/un-infested animals in the new location (thus spreading the disease/pest to the new location). This is achieved by modelling the movements at the individual agent's level, using sets of movement and geographical data. The agents that represent the premises have geographical point information associated within them which allows us to simulate the movement of livestock through the country by using the data obtained from the National Livestock Identification System (NLIS) for cattle and artificially generated movement data for sheep and pigs. Cattle identified with the NLIS devices are electronically scanned as they move through the livestock chain. At the time of scanning, each owner's PIC can be recorded and linked to the NLIS device. This transaction information is then stored in a secure, central NLIS database. Thus the NLIS database contains a complete record of all the locations an individual animal has been throughout the course of its lifetime. Movement information for other species is less readily available and this particular project relies on artificial movement data generated from expert's opinion and industry reports that outline management and selling behaviours. The details of the process by which this data is produced will be excluded as it lies outside the scope of this thesis. By cross referencing the movement data with the location information of each premise, the movement of individual animals (represented as the livestock agents) can be simulated throughout the country.

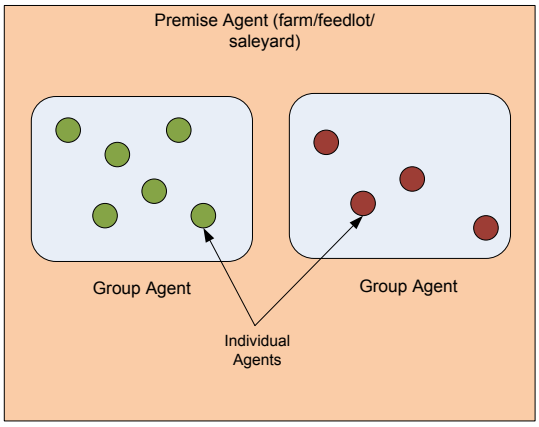

Figure 3.7 The agent hierarchy within the simulation. The premise agent can contain 0 or more groups of agents and these group agents can contain 1 or more livestock agents. In this example, the facility agent contains two groups of livestock agents.

The simulation models disease transmission within herds and between premises through epidemiologically important 'pathways'. The initial focus of the NMELDT has been to simulate outbreaks of foot and mouth disease (FMD). FMD is a highly contagious disease of livestock affecting cattle, sheep and goats, and pigs that would have a major economic impact were it to be introduced into Australia [75]. For FMD, in addition to simulating animal-to-animal spread within a herd, four pathways for spreading disease between premises have been included. In brief, the four pathways are:

- Direct contact spread. This pathway represents the spread of disease associated with movements of infected animals.
- Local Spread. This pathway refers to the spread of infection to neighbours and premises in close proximity to an infected premise where the actual source of the infection is not known and more than one possible mechanism can be identified. It has been recognised as particularly important for spread of diseases like FMD [76]. By definition, it could involve various methods including local aerosol spread across fences, movement of stock, vehicles, people, straying stock, run off, sharing of equipment between neighbours, etc.
- Windborne Spread. This pathway represents the disease being spread by airborne dispersal of the infectious agent by the wind. This is simulated using a range of climatic and biological data fed into a complex geospatial algorithm.
- Indirect Spread. This pathway incorporates a range of mechanisms by which diseases could be spread between premises, not involving live animals. It includes spread associated with contaminated products, equipment and inanimate objects as well as people and vehicles.

The NMELDT conceptually simulates the spread of disease based upon a *Susceptible (S), Exposed(E), Infectious(I), Recovered (or removed) (R)* (SEIR) disease lifecycle model, where an agent (livestock animal) progresses through the lifecycle of disease states outlined in (1):

$$Susceptible \rightarrow Exposed \rightarrow Infectious \rightarrow Recovered \quad (1)$$

In addition to the disease life cycle, a symptom life cycle is simulated that models the presence of clinical symptoms. This is separate, but related to the disease life cycle. The agent (livestock animal) progresses through the states of the symptom life cycle as outlined in (2):

$$Infected\ Without\ Symptoms \rightarrow Infected\ Showing\ Symptoms \rightarrow After\ symptoms \quad (2)$$

The disease states are:

1. *Susceptible*, where an animal is not infected with a disease but may become infected
2. *Exposed*, where the animal is infected but not yet infectious i.e. cannot pass the disease to others
3. *Infectious*, where the animal is infected and can pass the disease to other animals
4. *Recovered*, where the animal has recovered from the disease and is immune or is removed (e.g. if it dies of the disease or is culled). A recovered animal cannot be re-infected.

The symptom states are:

1. *Before symptoms*, where the animal may be infected but not showing symptoms
2. *Showing symptoms*, where the animal is infected and showing symptoms. Under some circumstances, a recovered animal may continue to show clinical evidence of the infection for a time.
3. *After Symptoms* where the animal has recovered and is no longer showing symptoms. Some infected animals may not show symptoms.

The agents maintain their own internal state information and update this information during discrete time cycles. A single simulation cycle represents a single day and each time cycle is broken down into four different stages. In each of these stages, different state update operations are carried out by agents. The simulation cycle consists of an early morning phase (00:00-06:00), a morning phase (06:00-12:00), an afternoon phase (12:00-18:00) and a night phase (18:00-24:00).The early morning phase is not used in the default setup. In the morning phase, livestock movements to saleyards are processed and a disease spread stage is carried out after the movements to represent the spread of disease within saleyards. In the afternoon phase, the movements away from saleyards and farm-to-farm direct (animal movements), indirect contacts and local spread pathways are processed. Finally, in the evening phase, a disease spread stage within the herd is carried out to represent the spread of disease between animals within an infected premise and windborne spread between different premises is simulated. Figure 3.8 outlines the simulation cycle.

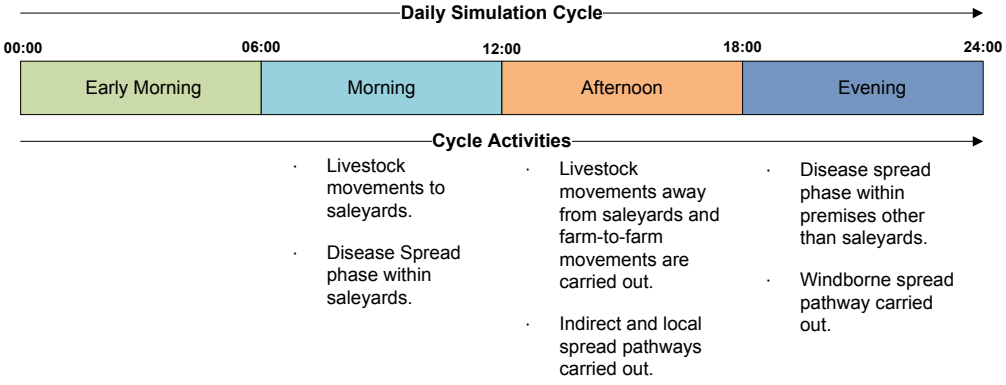

Figure 3.8 The conceptual simulation cycle used in the NMELDT model. This outlines the order of events within a single cycle of the model.

The individual livestock agents store 64 bits of state information that consists of a *disease natural history (DNH)* information and the agent's current state within this DNH. The disease natural history specifies the number of days that the animal spends in each stage of the Disease Lifecycle and the Symptom lifecycle. This scheme of states is represented using two 32 bit Java integers (totalling 64 bits per agent). One integer holds two indicators that store the current disease and symptom states and two counters that keep track of the number of cycles that the agent has spent in the current disease and symptomatic states. The second integer stores the numbers of days spent in each of the disease and symptom states (along with vaccination and disease carrier information). Figure 3.9, below, outlines this scheme.

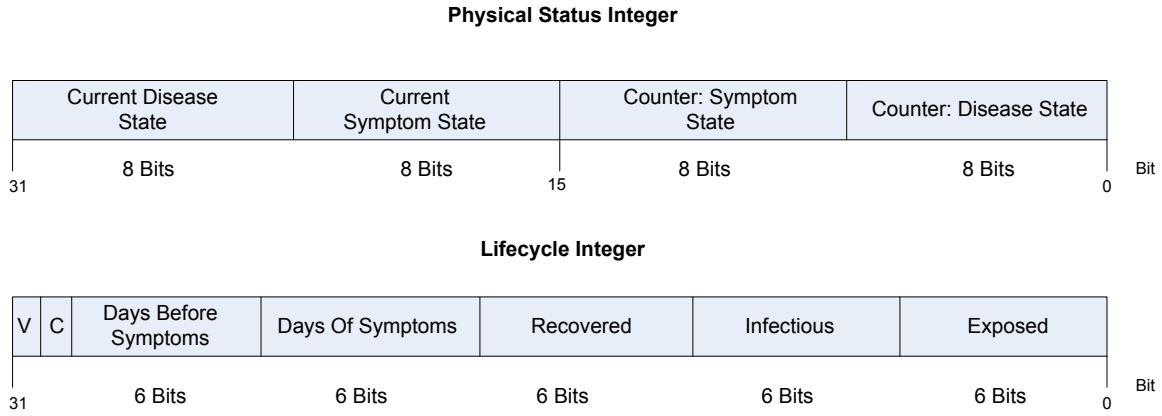

Figure 3.9 The agent's state information storage scheme

The life cycle information is generated stochastically from a set of distributions when the agent is created. When an agent is created, it is in the susceptible state and it remains static in this state until it becomes infected. Once infected, the agent transitions through its disease and symptomatic lifecycles using the counters in the physical status integer. Figure 3.10 and Figure 3.11 show the main interface of the model while running a sample scenario.

The data compression scheme proposed earlier in this chapter can be easily applied to the NMELDT, by treating the DNH and the physical status integers as separate data items that can be compressed across the

whole agent set. The individual data items within these integers are not extracted and compressed individually. These components are left grouped together and the integers storing these are treated as individual pieces of data for the purposes of the compression scheme. The result of this is that each agent effectively has two pieces of state information for compression. The first of these is the Disease Natural History (DNH). The data contained within this integer is static across the lifetime of the simulation, after the contents have been calculated when the agent that the data belongs to is initialised. Because the lifecycle information stored in the DNH is sampled from a set of distributions, there are a finite number of different combinations that can be stored in the set of components. As a result, there is significant redundancy of this data across the set of agents. This data is highly suitable for compression. The second data item contains the current physical and disease states. This data is dynamic, changing during in each cycle of the simulation. There is still a significant level of data redundancy as agents transitioning into the same disease state during the same cycle will have the same values stored in each counter. Based on these properties, the hierarchical scheme can be applied to compress this state information. Figure 3.12 gives a high level overview of the agent data hierarchy applied. The scheme uses two levels of abstraction to group agents that have identical information in the DNH integer and then within these groups, identical physical status are combined and a counter is maintained that tracks how many agents are represented by the same DNH/physical status set.

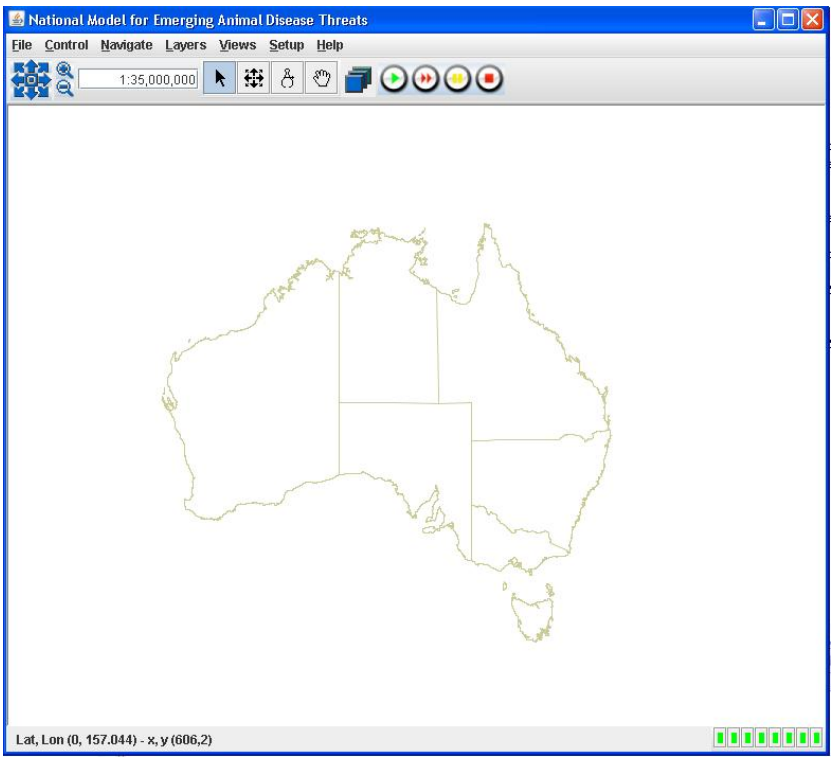

Figure 3.10 The main interface of the National Model for Emerging Livestock Disease Threats (NMELDT) system. The interface includes the OpenMap GIS component, displaying a map of Australia.

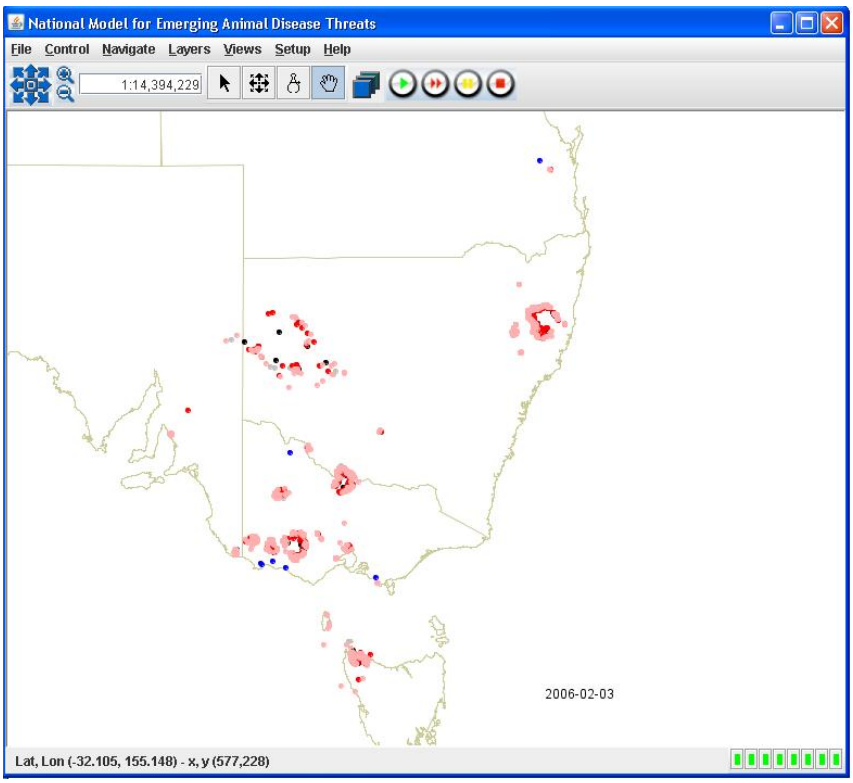

Figure 3.11 The National Model for Emerging Livestock Disease Threats system running a simulation. The coloured markers represent premises that contain livestock infected with FMD. The colour represents the state of the premises.

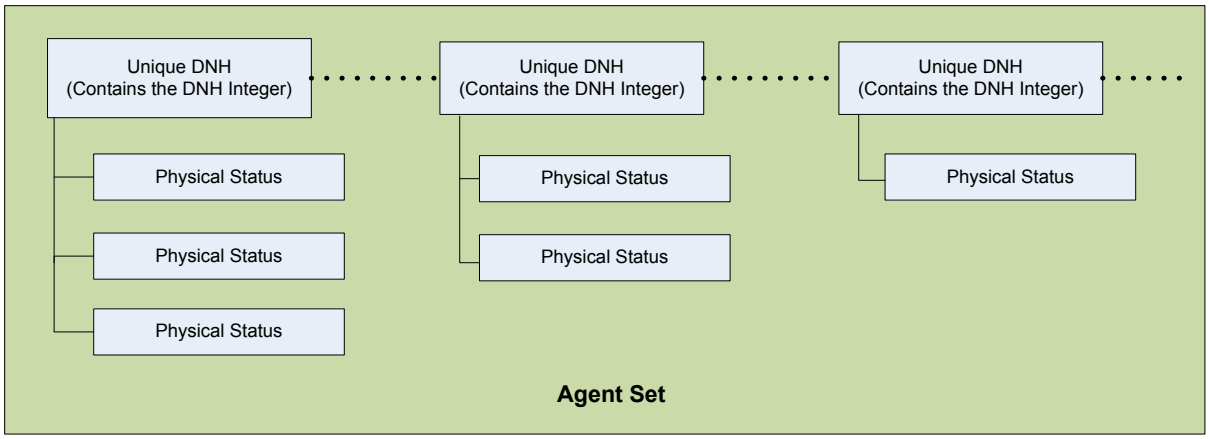

Figure 3.12 The agent data hierarchy scheme applied to the NMELDT.

The nature of the simulation also allows for the use of the optimised update algorithms to be implemented. Within this simulation, agents that are created or moved to the exposed state on the same logical day within the simulation, that have the same physical status and DNH will progress through the same set of states, changing at the same time up until they are removed from the simulation. This is a trivial update until the agent transitions to the exposed state. Similar to the susceptible state, agents have no state information changes. Agents that become exposed on the same day will progress through the remainder of their states together. As a result of this configuration, these agents can be updated without decompression in the manner described in the previous section. In simple terms, the current disease state, current symptom state along with their corresponding day counters can be updated within the hierarchy directly because the update operation

on the physical status/DNH combination that represents one or more logical agents will be identical across the logical agents it represents.

A complete analysis of the efficiency gains obtained from the application of the proposed Dynamic Agent Compression (DAC) scheme to NMELDT is not included in this thesis, but a simple experiment will be used to illustrate the reduction in the amount of memory used and the improvements in the efficiency of the update process. The experiment carried out involved a simulation loop that runs for 10 cycles. At each iteration of the loop 20,000 new agents were added with a 'physical status' value unique to that iteration. This represents that agents within a simulation that become infected at different times will have different physical states (e.g. all the agents that become infected on Day *y* will have the same physical status, but it will be different to that agents created on Day *x*) . Ten consecutive runs of the experiment were carried out using the DAC scheme with averages taken to insure validity. The results for this experiment are discussed below:

- Across the 10 simulation cycles 200,000 agents are created within the simulation scenario. This means that without DAC, 200,000 × 4 bytes are used for the Disease Natural History encoding, 200,000 × 4 bytes are used for the physical status encoding and 200,000 × 4 bytes are used for the agents pointers to their parent groups. This means that at the end of the tenth cycle, the agents occupy a total of 2,400,000 bytes or 2343.75Kb of memory.

- Using the DAC scheme, an average of 1,219 DNH level data items were created, to represent the Disease lifecycles and these 1,219 DNH data items contained a combined average total of 10,611 physical status data items that represent the physical statuses of all the logical agents generated in the simulation. This means that at the end of the tenth cycle, using DAC, 1219 × 4 bytes were used to store that agents Disease Natural Histories, 10,611 × 4 bytes were used to store the physical statuses of the agents, 10,611 × 4 bytes were used to store the physical status data and 200,000 × 4 bytes were used to store pointers to the parent groups. This means that the 200,000 agents in the simulation occupied a total space of 889,764 bytes or ≈868.9 Kb.

Based upon the above results using DAC to update all 200,000 agents present in the simulation at the end of the 10th cycle, it will only require 10,611 update operations. This is a major improvement over the implementation without the DAC, which would require 200,000 update operations (i.e. one update operation for each logical agent).

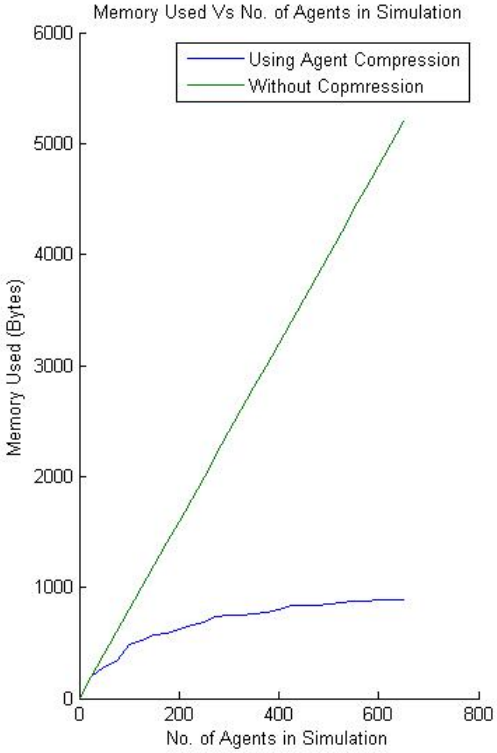

Figure 3.13 The memory usage as the number of agents is increased.

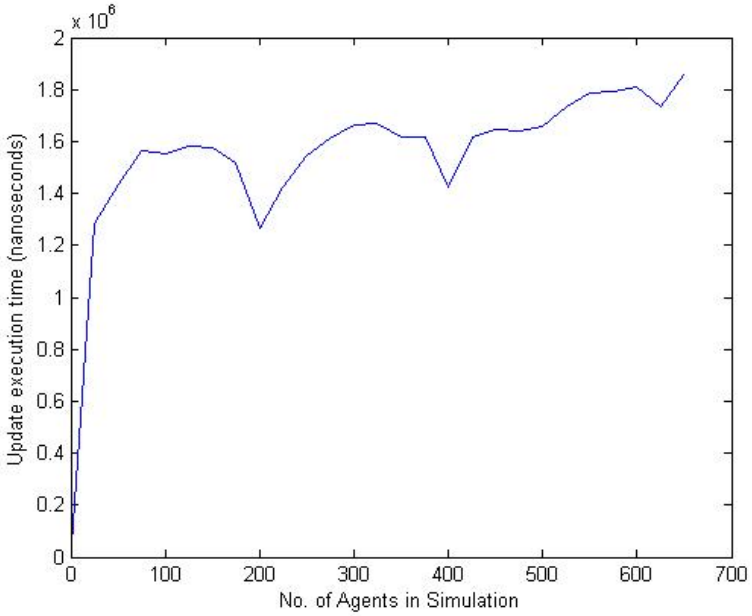

Figure 3.14 The execution time for the agent update process against increasing number of agents.

In order to compare the memory usage between a simulation using the DAC scheme and a simulation without DAC, simulations with increasing numbers of agents were run with different numbers of agents ranging from 25 to up to 650. Each simulation was run for a single cycle to insure that no 'new' agents were generated. At the end of each simulation run, the amount of memory used was analysed. Figure 3.13 shows the simulation system with compression uses significantly less memory than without the compression scheme applied. Further experimentation measured the execution time of the agent update process with increasing number of agents in the experimental simulation. Figure 3.14 confirmed that the update execution time rises by only 10% (i.e. from 1.6ns to 1.8ns) even when the number of agents increases 6-fold (i.e. from 100 to 600).

3.8 Concluding Remarks

This chapter has outlined the techniques for improving the processing efficiency and memory usage of large scale agent based modelling systems through the use of Dynamic Agent Compression. The work presented in this chapter shows how a data hierarchy, that represents all of the data items contained within all the agents present in a model, can take advantage of data redundancy to reduce the storage space required and the processing overhead in an ABM. A method for accessing the hierarchy through a centralised manager that allows the individual agent's data items to be used throughout the model was explained so that the scheme is usable in a wide variety of systems. The case study detailed a DAC's implementation for the *National Model for Emerging Livestock Disease Threats* (NMELDT) which provides a positive example of how compression techniques can be applied with merits to an existing modelling system. This implementation demonstrates the viability of the use of data compression within a real-world modelling system and how the techniques outlined for developing a compression hierarchy can be successfully applied to a set of agent data items. The experimental analysis shows the potential benefits that dynamic agent compression can provide in terms of memory savings and CPU clock cycles required for state update operations. The results show a saving of approximately 2/3 on the amount of memory required to store the agent set data in the final simulation cycle and a 50% decrease in the number of CPU operation required for updating the agent's state information. Further analyses show that the system scales well as the number of agents are increased within the simulation in terms of both execution time for update operations and memory usage.

The work completed in this chapter suggests a potential platform for implementing an agent-based model for OWSWF. However, additional analysis and development are required to fully understand the capabilities of this approach. There are two key reasons for this:

1. The requirements for the OWSWF modelling are very different from agent logic implemented within the NMELDT. The OWSWF lifecycle is far more detailed and the processing carried out is more intensive.
2. The OWSWF model has a spatial component that has not been addressed in the investigation so far.

In addition to the follow-on analysis required for the application of DAC to the agent-based OWSWF model, another area for development is the application of this scheme to heterogeneous agent systems. In the scheme outlined, the compression hierarchy can only be used to represent the data from a set of agents that are of the same type. If this scheme were applied to a heterogeneous simulation, a separate compression hierarchy and compression manager would need to be implemented for each type of agents in the system. A method of representing multiple agent types within a single compression hierarchy will allow the scheme to make use of data redundancy across different agent types. If we take the example from the previous sections of the agents that contained the *x* and *y* coordinate data and extend this to include a second class of agents that also have *x* and *y* coordinate data, these items can be compressed across the agent types. This could improve the performance of the compression hierarchy and reduce the overhead incurred by using multiple compression managers.

A possible approach for implementing the agent compression scheme across multiple agent types (i.e. with multiple hierarchies) is to apply multi-threaded, parallel processing techniques. In this approach, each agent type's compression will be managed by a separate set of threads that are responsible for the retrieval and insertion operations as described in the previous sections. This approach will support parallel updates to the agent's state data and limits the overhead presented by the additional hierarchies. This approach does, however, present its own problems such as the requirement of a multi-threaded environment (including the supporting hardware such as multi-core processors) to achieve the needed performance to overcome the overhead imposed by the data structure and the additional complexity due to memory management across the different structures of the model.