The final year project is described as an extensive piece of individual work relevant to the course. One of liberties this project offers is a chance to take a particular area of interest and significantly deepen the knowledge in the subject. Therefore, an important decision that is made early on in any final year project, is the choice of topic and subject areas on which to focus. An interest in the subject area can play a key part in a projects success or failure and for this reason the process of settling on a topic involved a significant amount of time and research.

The initial idea which was developed into a project proposal involved the production of software within the Natural Language Processing field. The primary aim was to implement an intelligent agent that would parse text, recognizing relationships between pronouns and the nouns they refer to. The program would store everything in a database that it could then parse in order to analyse the collected data statistically. Based on user feedback, as to whether the system was parsing correctly or not, the program would then self optimize the initial parsing rules given to it in order to parse more efficiently and accurately. Furthermore, this could be extended further to include text generation functionality allowing it to communicate with a user via a keyboard and screen. It would effectively function as a chat-bot which "learnt" from its mistakes via Artificial Intelligence algorithms such as reinforced learning and Genetic Algorithms. After further research into the Natural Language Processing field it was realized just how ambitious these plans were. The subject of data mining a corpus of descriptive text in isolation is a broad field subject to much academic research.

Following interest developed exploring the field of Artificial Intelligence in particular genetic algorithms a second concept took form. The aims were to recreate a race track and racing car and implement an intelligent agent with a limited view of the track such as a human would have. After exposing the functionality of the car to the agent it would then be given the goal of completing the track in the least amount of time possible. This idea was dismissed in favour of the final idea because it required more development on an environment and physics involved in simulating a car accurately than the development of an intelligent agent that it was possible that the Artificial Intelligence element would not get the time required to produce a finished artefact.

After reading around the field of Artificial Intelligence the topic of swarm intelligence was discovered which was first used in reference to robots. The definition given of Swarm Intelligence is "systems of non-intelligent robots exhibiting collectively intelligent behaviour evident in the ability to unpredictably produce 'specific' ([i.e.] not in a statistical sense) ordered patterns of matter in the external environment"[**?**] the idea of implementing a system that could simulate a group of simple objects that together worked intelligently was greatly appealing as a final year project. Ants and ant colonies provide an interesting application to the real world for the abstract idea behind swarm intelligence.

Also the large amount of ants often found in colonies offered the opportunity to look at the problem on a large scale, providing a large amount of computation in terms of decisions, movement and collisions.

During the process of drawing up the project proposal the decision was made to focus on how the problem could still be run efficiently while increasing the amount of computation being carried out. When reasoning about the problem it was clear that many decisions would need to be made at once, these circumstances would be best dealt with simultaneously, hence using parallel processing it would be possible to exploit this demand for things to be computed at the same time in order to efficiently process these simultaneous computations in parallel. As parallel processing would be an important factor in achieving an efficiency product it was decided that a range of programming paradigms should be researched in order to determine the most effective methods in achieving performance increases in this manner. The project proposal concluded as the production of an ant colony simulation with an emphasis on designing code, while exploring multiple paradigms, to produce a product that scaled as the problem size increased, through the use of parallel algorithms.

Relevance to Degree During the early stages of the project much reading was done on topics related to Artificial Intelligence, and the research in this area inspired the choice of project. This research also aided learning later in the course when CI342 - Advanced Artificial Intelligence was a required module. The project also covers the design of algorithms thus many of the topics covered in the CI312 - Computer Graphics Algorithms module provided valuable information in regard to measuring the time complexity of an algorithm. This module's assignment also encouraged the implementation of algorithms based on information presented in academic papers. The experience of successfully programming algorithms from this approach provided confidence when researching new techniques. Research techniques applied throughout the project were also developed in the module CI339 - Emerging Game Technologies, which involved an investigation and the production of an academic paper based on an emerging trend. CI346 - Programming Languages, Concurrency and Client-Server Computing was a module which not only greatly aided the study and development within this project but knowledge acquired throughout this process has also been of great assistance in the CI346 module. For instance, implementing a solution to a problem in the statically typed programming language ADA was far easier to realise than it would have been without the knowledge and experience gained within the project.

Major Aspects As identified in the preparation of the proposal there are two major aspects to this project, a simulation and multi-core programming. Simulations are sometimes seen as a genre of game; a life simulation game could revolve around a particular character and its relationships within the simulation, or it could be a simulation of an ecosystem [**?**]. Biological simulations often allow

the player to experiment with genetics, survival or ecosystems; this can often be for the purposes of education. Unlike other genres of games, simulation games do not usually have set goals or end conditions that allow a player to win the game. Rather, they focus on the experience of control whether it be the lives of people, when micromanaging a family (the most notable example of this is Will Wright's The Sims), to the overseeing the rise of a civilization or success of a business. Outside of recreational and educational games, biological simulations are utilized by researchers to test theories, which in practise would be impossible to carry out because of technological or financial constraints. It also gives the opportunity to show research findings visually. These are often large scale simulations which demand large amounts of processing power due to the nature of the problems. For example, neurological simulation in the Blue Brain Project[**?**] or weather prediction [**?**].

The second aspect of the project is multi-core programming - Processor clockspeeds aren't increasing at the rate they used to, with the focus of hardware manufacturers now producing processors consisting of multiple cores[**?**]. This shift in focus still keeps Moore's law intact, but order to harness the full potential of these multi-core processors, it is necessary to run computations on all of the processors' available cores. While it possible to describe the splitting up of work between a processor's cores as a trivial process, in practice it opens up several new issues which a programmer must address; into how many pieces should the computation be split, will the smaller computations need to communicate their results to other computations, is it necessary for some computations to finish executing before others begin? These are all questions which a programmer faces when designing an application to make use of a multi-core processor. There are two words which often occur in discussions about multi-core programming, Parallelism and Concurrency. It is important to clarify these terms when addressing the questions multi-core programming poses and further research will attempt to achieve this clarity.

Summary Although qualitative objectives were not achieved in their entirety the project has proven an academic success, as the knowledge gained and the potential for further work makes it a good candidate for continued research. The experiences gained over the course of the project now allows for the reasoning of new programming problems from a different perspective. Also the produced artefact, although incomplete has the potential to yield interesting results in the field of parallel processing. A detailed chronological and evaluative description of the implementation process can be found in Chapter 5 of this document. The following is a brief summary of the documents structure.

In Chapter 2 important concepts in parallelism and information from relevant publications on the topic is gathered. Studying them provided insights about the inherent problems in the development of large scale simulations and their possible solutions.

Chapter 3 is focused on specifying the goals of this project and on analysis of the algorithms that are going to be used as a basis of the application.

Chapter 4 details the evolution of the design for the final product explaining and justifying the architectural and design choices that were made.

Chapter 5 describes the progress made up to the current stage of the development process. Specific problems that appeared during implementation and testing and suggestions for their resolution are examined.

Chapter 6 shows and discusses the findings made and the level up to which the project goals were achieved. It also mentions possible improvements for the simulation and suggests new areas of investigation prompted by the lessons learned and the inevitable evolution of hardware.

Chapter 7 gives a brief summary of the points made in the previous chapters.

0.1 Research overview

In this project the problem of simulating the independent movements of a group of ants is tackled, Instead of taking a short-cut approach to the problem and allowing every ant to see the whole 'world' the ants view will be limited to more realistic sensory field. With each ant making decisions based on their own tasks, independently; while sharing the common goals of the colony. This will provide several problems which the project will attempt to address. Over the course of the project there will be a particular focus on how large the simulation can be while still running efficiently. This will take into account the amount of ants working individually yet simultaneously, while trying to keep the nature of their interactions as accurate as possible, as well as the size of the simulation world the ants are contained in.

It is important to mention, however, that the realistic behaviour of an ant and ant colony is not the aim of this project. Rather, the aim is to produce a simulation which exploits the parallelism offered by the ant colony problem. Therefore, a description of the behaviour of ants and the ant colony will be given below which should be agreeable with the average non-entomologist. After, there will be an analysis and discussion of various approaches to tackling the problem that has been posed, not only abstractly but programmatically. Firstly, looking at various programming paradigms, leading to a decision on a development language; before looking at different simulations which have tackled similar problems. Then research will pursue areas and methods that are uncovered through investigation, that provide potential solutions to the problem.

Ant Colony Generalizations Listed here are the aforementioned generalizations or preconceptions about ants and their colonies, which will be assumed for the purposes of this project. All ants need both food and water to survive, according to the Centre for Insect Science Education Outreach at the University of Arizona ants can go for quite some time without food but without water they will be dead within a day [**?**]. Therefore it will be expected that ants in order to survive will seek out food through exploratory strategies. Ants communicate through the use of pheromones, all have glands through which they can secrete

one or more types of pheromone[**?**]. Research has shown that ant pheromones can be used to communicate a range of messages but in general they are focused on marking paths to profitable food sources and signalling alarms or danger. An ant colony functions from single location called a nest, but this can take many shapes in the form of ant hills and underground colonies. In order to expand the colony it is necessary for resources to be brought back to the colony to facilitate the production of more worker ants. This need for resources can cause the expansion of the nest to slow, even though there are many ants working to expand it. Normally ant colonies size will number in the millions of workers before this is the case.[**?**].

To summarise, the generalised behaviour of an ant is an insect whose purpose is to locate for food and return with the food to its colonies nest. An ant will also communicate other food locations to other ants in the colony through the use of pheromones, during its limited lifetime which may be shortened by a lack of food. Where the behaviour of ant colony as a whole is concerned, the production rate of new ants is proportional to the amount of food successfully brought back to the colony. The physical size of the nest will be reflected by the amount of ants that belong to the colony.

Problem Area When the concept of this simulation is broken down it is evident that a crucial element of the system to be produced is the ant. As mentioned before, the amount of ants in a colony can grow to a very large number. Within this simulation however, in order to produce a working final product an incremental approach will be taken to solve the problem looking at first representing small amounts of ants. Then looking at producing functional collision detection between them, before increasing the complexity of an ant and increasing the amount of ants in the system. The project will be aiming to take an approach that scales well, so that when more ants are added to the simulation the performance does not drop to a point where it is no longer accurate.

The performance requirement is important to note when looking for a programming approach in the research stage the project, it would be sensible choose an approach that inherently produces solutions capable of scaling well, as the problem size increases so that performance does not dip when the demand on the system is increased. As highlighted in the project proposal the problem allows for an increase in performance by exploring the nature of parallelism within the problem. With this in mind a notable observation that can be made of the problem is that it is rich in concurrency. Concurrency is a property attributed to a system which performs more than one "possibly unrelated tasks at the same time."[**?**] Holding this definition against my problem, it is clear that this problem is a concurrent one when viewing ants as tasks.

Research Aims Throughout the following section a breakdown of the topics that will be important to the project will be provided. Questions will then be

derived from these topics, in order to produce a set of aims that the research will focus on addressing. When tackling a concurrent problem, it would profitable to research what are the most popular and promising programming paradigms used to approach parallel and concurrent problems; before taking the research further to look at why these paradigms are more efficient at dealing with such problems. Getting more specific, the focus then turns to what programming languages are used for this type of problem, again looking at why certain languages are preferred. Other important aspects of the project will also be further researched, such as collision detection, the visual representation of the simulation. Most importantly a programming paradigm will be chosen to focus on "the choice of programming paradigm can significantly influence the way one thinks about problems and expresses solutions" to problems. [**?**]

To summarise, the research aims of this project are as follows:

- Identify a programming paradigm that is suitable for the problem.
- Compare languages that could be used in the implementation of the project.
- Review algorithms and Data Structures (represented in languages which may be used throughout this project) that tackle concurrency and other problems found in my project.
- Analyse other large scale simulations.
- Look at approaches to large scale collision detection.
- Consider possible ways to represent the information in the simulation visually.

Programming Paradigms "Over the last decades, several programming paradigms emerged and profiled. The most important ones are: imperative, object-oriented, functional, and logic paradigm."[**?**] The next few paragraphs will look briefly at these four paradigms and discuss their strengths and weakness, keeping the multi-core programming element of the proposed problem in mind.

The Imperative Paradigm The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940s. [**?**] This means the programming paradigm, similar to the Von Neumann machine, operates by performing one operation at a time, on certain pieces of data retrieved from memory, in sequential order. According to Backus [**?**] the man who coined the term "The von Neumann bottleneck", "there are several problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control." The essence of the Von Neumann architecture is the concept of a modifiable storage. Variables and assignments are the programming language representation of this modifiable storage. This is then is manipulated by the program's code, just as the

variables and assignment statements within the program dictate. Imperative programming languages provide a variety of commands along with these assignment statements to provide structure to code and to manipulate the store.[**?**]

It can be said that an imperative programming language defines a particular interface through which the programmer can view the hardware. The real reasoning as to how the program's code is interpreted is left to the compiler. Speaking on the design of the early imperative programming language Fortran Backhus remarks " we simply made up the language as we went along. We did not regard language design as a difcult problem, merely a simple prelude to the real problem: designing a compiler which could produce efcient programs"[**?**].

The Object Orientated Paradigm The object orientated paradigm focuses on, as its name suggests, elements of a program which it calls objects. Objects are groups of similar data and functionality related to that data held by the object. The data and functions which are grouped within an object can then have their access restricted from functions and data contained in other objects. This process is commonly called encapsulation and is common in the Object orientated paradigm [**?**]. Encapsulation allows for a more formal structuring of a program than imperative languages. However, most object orientated languages still have mutable state within each object. The way in which this mutable state is manipulated is often very similar to that of imperative languages[**?**].

As well as the concept of encapsulation object orientated languages introduce other concepts such as Abstraction and Inheritance which in order to implement, encourage the programmer to look for areas where the code they are writing could be reused [**?**]. The modular structure of code which results from the correct use of Object orientated programming techniques means that these reusable modules can also be tested separately. Allowing for tests to be modules focused can result in a greater amount of more specific tests which are more likely to catch errors in code.

When looking at research done into the benefits of programming in these paradigms from a parallel perspective a study into speculative module-level parallelism in both the imperative and object oriented paradigms showed no benefits with tests being run in several parallel algorithms developed in both the Java and C programming languages. It was highlighted at the beginning of the study that data dependences between threads (on return or memory values) would limit the speed-up obtained. But the encapsulation of data as an effect of an object-orientated language would favour better performance in the Java programs. Interestingly this was not the case [**?**].

When analysing the thread based approaches imperative languages provide through POSIX threads (pthreads) in C++'s support libraries[**?**] and Java's implementation of pthreads on the Java Virtual Machine [**?**]. It came to light

that caution must be taken in use of multiple threads in larger applications "use of threads can be very deceptive ... in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible." [**?**]. When a program running several multiple threads which interact with the same data structure it is difficult to keep the results of computations deterministic, as this relies on each thread being guaranteed a mutable data structure is coherent when accessing and updating it. In an article describing these issues in the programming language Java the warning is given "Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism ... this is particularly true in Java ... we urge you to think twice about using threads in cases where they are not absolutely necessary. [**?**]"

The Functional Paradigm Object orientated programming has been described as "the antithesis of functional programming" [**?**] From the two programming paradigms above, it can be deduced that a good programming language is modular. But John Hughes claims that it is not enough for a language to just support modularity, it needs to go a step further and make modular programming easy. To do this, a programming language needs to provide flexible functionality, in order to bring modules together. In an article on Functional Programming Hughes writes "Functional programming languages provide two new kinds of glue - higher-order functions and lazy evaluation."[**?**]

Functions play an important role in functional programming languages and are considered to be values just like integers or strings. A function can return another function, it can take a function as a parameter, and they can even be constructed by composing more than one function together. This offers a stronger "glue" to combine the modules of a program. The title 'higher order' is the title given to functions which follow the lambda calculus representation of functions[**?**]. In lambda calculus functions may be partially applied to allow for the creation of new functions which hold the values of the parameters already provided. Functions are also the source of program control and flow in this paradigm. By calling further functions within themselves functions can specify a sequence of operations and by calling a function from within its self the programmer can achieve iteration through recursion.

In a functional setting there is no state, the hardware related model of viewing memory as containers is abstracted away. Which means that the concept of a variable is quite different in the functional paradigm. What might appear to be an assignment statement in an imperative language, in a functional language translates to a name becoming bound to a value. The way variables are treated in functional languages is similar to the way variables are treated in mathematics (once x is assigned a value during a calculation x is always that value). When data is assigned to structures in this way the structures are called immutable [**?**]. Modifying such data structures therefore creates new data in computer memory maintaining the existence of the old data structure. This

Characteristic	Imperative approach	Function
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What in
State changes	Important.	Non-exis
Order of execution	Important.	Low imp
Primary flow control	Loops, conditionals, and function (method) calls.	Function
Primary manipulation unit	Instances of structures or classes.	Function

Table 1: Table of Foos

process remains efficient however as a whole data structure isn't reproduced for a small change, as references back to the original data are made where no changes occur[**?**]. Immutability can therefore aid in reasoning about parallelism and keeping data coherent as read-only data structures can be passed to parallel processes and modifications that occur leave the old structure still intact.

[diagram table of differences] Highlighting the differences between the Functional paradigm [**?**]

The Logical Paradigm Logic programming is characterized by its use of relations and inference.[**?**] The logic programming paradigm influences have led to the creation of deductive databases which enhance relational databases by providing deduction capabilities. The logic programming paradigm can be summarised with its three main features. Firstly computation occurs over a domain of terms defined in a "universal alphabet". Secondly, values are assigned to variables through automatic substitutions called "most general unifiers", in some situations these assigned values may themselves be variables and are called logical variables [**?**]. Finally the control of a program in the logical paradigm comes from backtracking alone. Here lie the reasons for both the strengths and weaknesses of the logical paradigm. Its strengths come in the form of great simplicity and conciseness, while its weakness lies in having only "one control mechanism" and "a single data type" [**?**].

Development Language Based on the research into the benefits of each programming paradigm, it appears that the functional programming paradigm has more to offer a developer looking to produce a solution for a parallel problem. In particular, the Functional paradigm works differently to the other paradigms, due to its ability to abstract away from the sequencing that is normally put in place on functions programmed in the Imperative and Object Orientated paradigms. Their ability to be reasoned about mathematically is also a strength that will allow code developed to be analysed for correctness. When functions in the functional programming paradigm are pure there are several ways to evaluate expressions in a parallel manner.

Erlang Joe Armstrong, when talking about the language that he created, has emphasised that robustness is a key aim of Erlang [**?**]. The language was

developed by the telecommunications company Ericsson during a pursuit for a better way of approaching programming. Safety and error management are very important in environments such as telecommunications, as downtime is something to be avoided at all costs. Such is the focus on error handling in Erlang that there are three kinds of exceptions; Errors, Throws and Exits, these all have different uses within programs [**?**]. Proper use of Erlang's extensive error handling systems can result in programming the robust applications Armstrong was aiming for. Another interesting and novel feature of Erlang is its support for continuous operation. The language has primitives which allow code to be replaced in a running system, allowing different versions of the code to execute at the same time [**?**]. This is extremely beneficial for systems which ideally should not stop running such as, telephone exchanges or air traffic control systems.

Memory in Erlang, is allocated automatically when required, and deallocated when no longer used. So typical programming errors caused by bad memory management will not occur. This is because Erlang is a memory managed language with a real-time garbage collector [**?**]. Erlang was initially implemented in Prolog which may explain its declarative syntax and its use of pattern matching[**?**]. The language also has a dynamic type system, meaning the majority of its type checking is performed at run-time as opposed to during compile-time.

This means errors the programmer makes, concerning variable types, may occur at runtime, possibly quite distant from the place where the programming mistake was made. Although dynamic typing may make it easier to get code compiling in the first instance, it may also make bugs difficult to locate later on in the development process. Erlang has a process-based model of concurrency using asynchronous message passing. The concurrency mechanisms in Erlang are processes which require little memory (lightweight), and creating and deleting processes and message passing require little computational effort [**?**]. These benefits are due in part to the fact that Erlang is intended for programming soft real-time systems where response times are required to be within milliseconds.

With its strengths in reliability, error handling and light weight concurrency, Erlang naturally excels at distributive programming. Erlang has no shared memory,

Haskell Haskell is a purely functional programming language, this means a function will always return the same result if passed the same parameters; a function such as this is also known as having the property of determinism. Keeping code pure (purely functional) eradicates the possibility of many bugs which find their cause from a dependence on side effects the programmer assumed would be there but were not. This kind of issue is increasingly likely to happen the larger an impure program grows, but Haskell is free of this problem. Determinism is not only of value to the programmer but the compiler can make use of

a property called referential transparency, a by-product of determinism. This allows the compiler to in reason about the program by "substituting equals for equals", Haskell maintains this referential transparency even though it is possible to produce input output and other sorts of computations which appear to have side effects, by encapsulating actions which cause side effects within a monad [**?**].

Garbage collection is also featured in Haskell[**?**] as in Erlang, this abstraction away from resource management allows the programmer the freedom of not having to specifically allocate and deallocate memory, which done wrong leads to several problems. But Haskell continues this abstraction to another fundamental area of programming, that of sequencing. Due to the effects of purity the functions order of execution is flexible while the program still yields the same results. Deterministic programs also make it easier to reason about parallelism, as a programmer no longer has to strip away unnecessary non-determinism [**?**].

The modularity of Haskell allows us to define generic functions, so functionality can be passed as an argument to higher order functions. This allows for more sophisticated functions to be written. For example, a function which traverses a data structure may be combined with a function which conditionally increments a value, producing a function which updates a data structure in a concise yet flexible way. The concept of conciseness and elegance is often discussed within the Haskell community.

Non-strictness or 'laziness' is another feature of Haskell, meaning that nothing is evaluated until it has to be evaluated. This allows for effective use of the functional operation filter, which allows the programmer to generate solutions which when given a set of all possible solutions extract only the values which are valid solutions. Non-strict evaluation ensures that only the minimum amount of computation is carried out. This also allows for infinite lists to be made use of, as only the values of the list that are need are computed and only when necessary [**?**].

Haskell is a strongly typed language, therefore it is impossible to implicitly convert between types. This means that all type conversions in Haskell program must be explicitly called for by the programmer. Forcing the programmer to be upfront about the types being used in the program allows for the detection and prevention of bugs, where the compiler may have made the wrong decision concerning the type of a variable in memory. Haskell's type system can be utilized through the use of type signatures which allow the programmer to express the type of parameters the function should expect and the type of the value the function will return. Haskell makes use of the Hindley-Milner type inferencing system allowing Haskell to deduce the types when a programmer does not use type signatures.[**?**] In these cases the inference system treats the value as the most general type it can. Another advantage brought to the programmer by

Haskell's type system is that of static typing. A type checker confirms that the types of functions results and parameters match up correctly so that there are no type mismatches at runtime. With such rigid checks in place before the program compiles, resulting in the majority of compiled programs always running as expected.

Scala Scala is a modern multi-paradigm programming language that "fuses object-oriented and functional programming" paradigms [**?**]. Similarly to Haskell, Scala is statically typed so the programmer can take advantage of the fact that once their program compiles, it will be virtually free of runtime type errors. Scala is also a functional language in the sense that every function is a value [**?**]. The language also allows for common functional features such as the use of higher-order functions, nested functions, and supports currying. Currying is an approach to dealing with functions, which was discovered by Moses Schnfinkel and later re-discovered by Haskell Curry, where all functions must accept a single argument [**?**]. This allows for the partial application of functions - a feature also found in the programming language Haskell. Partial application lets the programmer "avoid writing tiresome throwaway functions"[**?**] as functionality of a function with not all of its parameters sufficed will operate with the same functionality once the final parameter is supplied.

Scala evangelists describe the language as being purely Object Oriented, as "every value is an object" [**?**] and the language provides a very powerful way to inherit from multiple classes, even when more than one class defines functionality for a particular function. This feature is called mixin-based composition. Even though Scala is functional, because every function is a value (this means functions are also objects), it is not pure. Scala allows for assignment in order to incorporate some of its other features and therefore side effects are allowed. As a result, referential transparency is lost, which prevents the programmer reasoning about the structure of the code in a mathematical way[**?**].

An interesting feature which Scala delivers is interoperability with Java. In order to achieve this a few compromises were made[**?**], so that the languages are compatible in both directions, allowing the Scala compiler to compile Java code and vice versa. This feature may also be viewed negatively as it can cause a programmer to lapse back into standard Object Orientated or even imperative programming practices. Although cross compatible, some features of Scala are not natively supported by the Java Virtual Machine. For example the Java virtual machine lacks primitives in order to make the tail call optimization that allows for the efficient use of recursion. This leads to Scala code sometimes compiling to form a larger amount of byte and sometimes running slower than the equivalent written using Java's loop constructs [**?**].

Summary and choice This project chose Haskell as the development language for the solution even though it had not been encountered prior to the

initial research of this project. As such, the language was learnt throughout the duration of the project because of the substantial amount of benefits the language provides in tackling the problem. It offers referential transparency through its purity, allowing parallelism in algorithms to be exploited in an easier manner than would be possible in imperative languages. Of the languages that have been reviewed here Haskell's type system is the richest and most expressive. Haskell is also the strictest on purity. While Erlang maintains purity within its processes, Haskell forces the programmer to separate code with side effects from pure code with the use of monads. In contrast, Scala embraces impurity, allowing the programmer to freely choose whether their functions should be side affect free or not. The higher levels of abstraction offered by Haskell promote it further as a language, which would incubate design ideas for new and more efficient algorithms.

The Haskell compiler GHC supports multi-core programming as installed and includes multiple parallel programming models: Strategies [**?**], Concurrent Haskell [**?**] with Software Transactional Memory [**?**], and Data Parallel Haskell [**?**]. Furthermore, it has many third party libraries for distributed parallel frameworks such as MapReduce and the Actor model, in the 'compressed' and 'remote' packages respectively. This extensive support for a variety of multi-core programming techniques is unrivalled by either Scala or Erlang.

The speed of Haskell is also something that makes Haskell an attractive choice when looking to choose a development language for the project. In 'The Great Programming Shootout', Haskell comes close to matching the performance of optimised imperative languages as shown in the figure above. However, it is important to note that this comparison of language speeds does not accurately reflect the true speed of each language, as it is affected by the developers' knowledge of both their language and the algorithms the tests are based on. However, these graphs do shed some light on the performance times that could be achieved in each given language.

Approaches to Parallelism There are known models which programmers may base a parallel algorithm around and some lend themselves better to particular problems than others. Indeed, some like the Actor model or dynamic multi threading have already been mentioned in passing during the discussion focused on a suitable development language. In the following section some of the approaches taken in order to achieve parallelism will be looked at in greater detail, so that more informed decisions could be made during the program design phase of this project. Before looking at these methods, some of the theory behind parallel processing will be highlighted to show what can be expected from a parallel algorithm in terms of performance and scalability.

The Limits of Parallelization It has been proved by Gene Amdahl that increasing the amount of processors working on a program only increases speed for so long. Amdahl's law shows that the speed-up of a program is limited by the sequential portion of the program. This is clear because sequential processes

need to be performed in sequence so by definition they are prevented from benefiting from multiple processors. In 1967 at the AFIPS Spring joint Computer Conference, while talking about the overhead of "data management housekeeping" which can be found in any computer program, Amdahl reportedly said that "The nature of this overhead appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput."[**?**] This upper limit has been deduced from his conference talk and given rise to the widely used formula below.

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{1}$$

The formula calculates the potential speed-up that could be brought to a program by adding more processors work on the parallel portion of a problem. The sequential overhead generated in arranging the parallel computation that Amdahl mentions is represented by (1-P) and therefore P representing the parallel section of the program makes up the whole. The formula then takes the sequential portion and adds it to the parallel portion of the program over N, which represents the number of processors. This symbolizes the division of the parallel portion of the program between the processors. The upper limit that Amdahl refers to is clear as you increase the number of processors N, if you had infinity processors working on the parallel part the speed up would simply be 1 over (1-P). [**?**]

Software Transactional Memory The approach taken in Software Transactional Memory (STM) can be seen as Task parallelism in this approach the tasks run on separate threads which have to be explicitly invoked by the programmer. Also tasks may need to communicate with either tasks and to achieve deterministic result they must be synchronised via locks or messages. With this approach a programmer can expect to see modest amounts of parallelism and "as the number of CPUs increases, it becomes more and more difcult for a programmer to deal with the interactions of large numbers of threads"[**?**]. STM provides a different approach to these synchronisation issues than those found in conventional multi threaded programming, by providing a software abstraction of the transactional memory technique used on multi-core processors. This is achieved in the language Haskell with the use of atomic blocks [**?**]. Within an atomic block a snapshot of memory is taken and the operations within the block are carried out, when this block of memory is read checks are carried out to determine if the changes within the atomic block leave all other atomic blocks with the same view of memory. Thus, the program can guarantee the isolation of any changes that are performed within an atomic block and that the effects of a block become visible to another thread all at once; preventing a process from seeing a temporary view of memory.

Data Parallelism The approach taken by Data parallelism is to apply an operation simultaneously on a bulk of data. This allows for potentially massive amounts of parallelism as the amount computed in parallel corresponds to the amount of data you have to compute. In contrast with Task parallelism this is far easier to program. In a language free of side effects the bulk of the computation could be achieved with the following lines of code.

foreach i in 1..N ...do somePureFunction to A[i]...

Focusing the parallel algorithm on data in this way rather than tasks results in a single flow of control and synchronisation is not an issue each item of data in A is being passed to a pure function with no side effects (where A is some data structure and i is a valid index of that structure). Research and development is currently being undertaken for an extension on this approach in Haskell under the name Nested Data parallelism [**?**]. This would allow for 'somePureFunction' to also contain parallelism, introducing further complexity to the approach. Nested Data Parallelism would create a graph of parallel processes which, when taking into account conditional statements, would also create further load balancing issues for the compiler to handle, work on achieving automatic optimal performance in nested data parallelism is still under development.

The Actor Model The Actor Model is based a message passing system between units of computation which are called actors. "Actors are basically concurrent processes which communicate through asynchronous message passing" [**?**]. Message passing actors can create new actors, decide what action to take based on a message they currently have or send a message to another actor, all simultaneous to receiving a message. A model is implemented in both the programming languages Erlang and Scala. Implementations of the Actor model can be divided into two groups - firstly Actor languages which are Programming languages providing native actor support. Operations to send and receive messages are part of the programming languages implementation and as such are supported and integrated from the bottom-up. Secondly, Actor Frameworks, libraries, which include the Scala Actors library or Haskell's remote package, implement the model for an existing programming language. The performance of an Actor framework relies heavily upon its foundation. In Scala's case it is the Java Virtual Machine, the Virtual Machine provides concurrency based on Java's threading model.

Erlang, as an actor language, has threads which are very lightweight [**?**]. During runtime every actor is a thread of its own, which is possible due to Erlang's efficient context switching. Scala however, is reliant upon the Java Virtual Machine's threads which are heavy in comparison to Erlang, having time-consuming context switches and high memory consumption[**?**]. This places limits on Scala Actors at their foundation as the programmer now has to make a choice which could increase complexity. Either allow a thread to be assigned exclusively to an actor or not, both approaches hampering a programs ability to scale well with respect to the number of actors.

The Consumer Producer Model Another common concurrent programming paradigm where a process 'consumes' data from one or more data structures, produced or updated by a second process which runs either concurrently or in parallel. The data structure that is used by the two processes can be seen as a buffer that the producer fills and the consumer empties [**?**]. Parallelism can be achieved by having multiple producers or consumers so that while a producer process is preparing the next piece of data to append to the buffer another producer process can update the buffer. In this parallel case where multiple multiple processes are making an update to the buffer a thread safe implementation requires the buffer to be synchronized to ensure the buffers coherence.

If the buffer data structure is a fixed length, producers have to halt when the buffer is full. This effect can achieved via applying a barrier condition to the function which adds to the buffer and can be checked at regular intervals. A more sophisticated approach often seen in the use of Java's threads is to 'sleep' the thread and 'notify' the sleeping thread that it can resume producing once a consumer has made room in the buffer by removing an element. Sleep can also be called on the consumer threads when the buffer to be empty. When the producer updates the buffer with an element of data, it notifies the sleeping consumer process. Care must be taken over the management of threads in this fashion as deadlock could easily occur if a consumer thread is not notified that the buffer contains data that is to be removed.

Divide and Conquer The divide and conquer approach involves the decomposition of a problem into smaller subproblems. This approach can improve program modularity, and often produce simple and efficient algorithms. It can therefore be a useful tool in designing both sequential and parallel algorithms. Due to the need for subproblems to be independent there is often inherent parallelism even in a sequential divide and conquer algorithm. The smaller subproblems once solved can be merged together to compute the final result. In order for divide and conquer algorithms to achieve the most parallelism, it is often necessary to also parallelize the divide step and the merge steps.

The MapReduce architecture is an application of the divide and conquer technique. However, useful implementations of the MapReduce architecture will have many other features in place to efficiently 'divide', 'conquer', and finally 'reduce' the problem set. A large industrial implementation of MapReduce could incorporate thousands of compute nodes, the steps required to partition the work, run the computation and then finally collect all results quickly become non-trivial. On larger scale issues can arise such as load balancing and dead node detection which are hard problems in and of themselves [**?**]. While additional features maybe advantageous or even become a requirement such as saving interim state to recuperate from problems which may occur during a computation which runs for an extended period of time.

Embarassingly Parallel Embarrassingly Parallel, is the term given to problems where tasks can be performed in parallel with no need to reduce the computations performed by each process to one single result. Due to these problems having no dependence on each other for computation and do not need to be combined in any way. They are the easiest problems to parallelize and therefore it is important to recognise these situations and capitalise on them; possibly taking measures to introduce the opportunity for embarrassing parallelisation.

Large Scale Simulations Before creating a large scale simulation it would be advisable to look at how others have approached the problem of simulating large amounts of data in the past. Doing so may provide insights into techniques used, and shed light on problems that maybe encountered during the implementation process that maybe countered or even avoided during the design phase of this project. In this section a few large scale systems will be looked at describing what they do and looking at what steps were or may have been taken to produce an effective simulation.

Networks simulations In a study on simulation computer networks[**?**], the transmission and queuing of packets is simulated in order to evaluate and design network protocols. In such simulations the amount of memory limits the amount of nodes that can be simulated as the state of each node must be stored and the runtime is affected by the amount of network traffic. Two approaches are seen to approaching parallelism in this context. Firstly, a time parallel approach which requires splitting the problem based on time and assigning a processor to simulate the system over an assigned time interval. The study reports this approach can cause an increase in the execution speed of simulations enabling more network traffic to be simulated in real time. But does not allow for an increase in the size of the network being simulated. The second method takes a space-parallel approach by partitioning a large network, the nodes within each partition are then mapped to a different processor. This parallel technique introduces the parallel speed-up seen in the time parallel approach while allowing for the network size to increase as shown in the figure below.

Molecular dynamics Simulations in the field of Molecular Dynamics focus on the interactions between molecules as they are subjected to Newton's equations of motion. In a paper on parallel simulations with the Molecular Dynamics field, three parallel algorithms are used to solve the problem before being analysed and compared. The rst assigned each processor a xed subset of the atoms in the simulation to process; the second approach assigned a xed subset of the force calculations, at a given simulation step, to each processor and the third assigned each processor the computation of a xed spatial region [**?**]. Each technique applied to the problem displayed different advantages and disadvantages.

By partitioning the problem with respect to the atoms was found to be the simplest way of implementing the problem and had an added advantage of

loadbalancing automatically, keeping all processors working the same amount. Dividing the problem according to the force calculations is reportedly relatively simple and has the advantage of load-balancing similar to partitioning b the problem by atoms, with spatial partitioning being the hardest method with which to achieve load balancing. In terms of scaling with the problem it was shown that spatial decomposition achieved optimal performance. While dividing the problem with respect to the force calculations performed better than dividing by atoms because of the communication costs of having all the processors update each other on the state of their atoms [**?**].

Simulations for Games and Other Purposes Colony is crowd simulation, produced by Intel, which is optimized to run in parallel making use of the 'Thread Building Blocks' library that Intel develop and distribute [**?**]. In this approach a tool was used to analyse the performance of the processors threads and measure their effectiveness in reducing the run time of the problem. When setting out on designing this simulation the developers identified dependencies of tasksets [**?**]. These are groups of tasks which must complete before a new taskset can begin execution. The simulation sequence was then partitioned in manner so that each taskset did not require data to be calculated in other tasks within the set, and that the range of data in any particular set can easily be split up before being operated on.

The final simulation loop for Colony performed three steps in parallel, operating on tasksets, stored in arrays where the start and end indices indicated a group of tasks. Processing of these arrays could happen in parallel across the cores of the CPU in a Single Instruction Multiple Data manner [**?**]. Firstly, units within the simulation were grouped into 'bins', groups that could be processed in parallel. Secondly, new directions were determined for the units in each bin (this step depends the first). Finally, the units were updated following their new direction any other unit logic was processed (this step depends on the second) [**?**]. The developers of this simulation regard the identifying blocks of sequential code with the use of runtime performance analysis tools and storing data so that it can be easily broken up into chunks for processing; are the key to easily achieving parallelism.

Collision Detection In a large scale simulation in which bodies can collide, it follows naturally that in denser areas of the simulation there will be large amounts of potential collisions to detect. In order to make an informed decision on how to best tackle the collision detection problem, research was carried out on various areas of collision detection and methods that may allow for parallelism. Approaches to collision detection can be split into roughly two groups, static and dynamic. Static collision detection involves detecting the intersection between objects, at discrete points in time, during their motion [**?**]. At each of these points in time the objects velocities are disregarded, the tests carried out only check to see whether there is intersection between the geometries at the given point in time. Dynamic collision detection considers the continuous motion of

the objects over a given time interval taking to account the velocities of each object. Thus, static collision detection can only detect collision after it has happened, while dynamic collision detection all though requiring more to be computed about a given scenario will allow for detection of a collision before geometries intersect.

Bounding Volumes Bounding volumes can be used to simplify collision detection on complex objects, or can be used as a optimization stage to prevent unnecessary complex checks being carried out as often. This is achieved by surrounding the original object in a geometry that is easy to compute and allows inexpensive intersection tests, while using little extra memory [**?**]. Additional benefits can be obtained from a bounding volume when it is tight fitting to the original object and it is easily rotated or transformed. In order to provide inexpensive intersection tests, the bounding volume should have a simple geometric shape, however a compromise on simplicity may have to be made in order for it to be tight fitting. The better the fit the more the bounding volume can filter the need for unnecessary more expensive and accurate collision tests. The Figure taken from [**?**] below, illustrates some of the trade-offs between the five most common bounding volume types.

Partitioning Spatial partitioning techniques offer an even broader phase of collision detection than bounding volumes by dividing space into regions and testing if objects occur in the same region of space. As no intersection is possible if objects do not share the same region of space. When a space is partitioned correctly, the amount of checks between pairs of objects can be reduced as these checks will only have to occur between objects in the same partition. Navigation meshes are one method of partitioning a space in order to optimize collision detection. This optimizes collision by overlaying the world with a grid, the collision detection system would then keep track of which objects are in which grid cells, or this can be calculated based on an object's location. Once the grid cell of an object is determined collision checks only need to occur between objects occupying the same grid cell.

Binary Space Partitioning is another method of dividing the space in which collision detection takes place for the purpose of optimisation. An n dimensional space is divided by and (n-1) dimensional partition, for example a 3D volume is cut in two by a 2D plane, or a 2D plane is divided in two by a line. Objects falling in each division of the plane would then be stored into separate nodes of a tree structure with the root of these two nodes represented the divided plane. Further divisions can then take place to partition the two divided planes creating further leaves and nodes in the tree. Collision checks then can be limited to objects which appear within the same partition on the premise that an object needs to be in the same partition in order to collide with another object. k-d Tees are a restricted form of Binary Space Partitioning as the partitions made are aligned to an axis. They also hold the objects partitioned in a tree structure.

The advantage of using tree structures is that they can be built efficiently on average in O(n log n) time [**?**]. Also as objects located in partitions near to the object can be identified quickly via accessing the surrounding parent and child nodes in O(log n), the nearest neighbour problem can be solved more practically; eliminating the need to query the location of all the objects in the simulation. A benefit that be gained from spatial partitioning that has been identified in the analysis of other parallel simulations is that the collision checks within each partition can be run in parallel providing no objects overlap partitions.

Visual Representation Although development and testing will primarily work with a textual representation of the information within the system, it would be valuable to produce a more visually appealing representation of the simulation allowing the information presented to be comprehended easily.

OpenGL 2D

Definitions

Social, Ethical and Legal Issues Before embarking on this project it was essential to look into any social, ethical or legal issues that may arise throughout the course of the project or as a result of using the project's delieverable. It was noted that care must be taken when using or referring to material that influenced the project especially throughout its research phase. Citations have been made crediting such materials as necessary a full list of which can be found in the bibliography.

Legal- library licensing - ensure that the usage of the product conforms to the licenses of the libraries used. A discussion of the licences that each library is provided under.

The utility-ht package Various small helper functions for Lists, Maybes, Tuples, Functions. Some of these functions are improved implementations of standard functions. They have the same name as their standard counterparts. The package only contains functions that do not require packages other than the base package. Thus you do not risk a dependency avalanche by importing it. However, further splitting the base package might invalidate this statement. License BSD3

The GLUT package A Haskell binding for the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. For more information about the C library on which this binding is based, please see: http://www.opengl.org/resources/libraries/glut/. License BSD3

The QuickCheck package QuickCheck is a library for random testing of program properties. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators. License BSD3

Also the base libraries in Haskell are licensed under a BSD style license, both these licenses state that the BSD license must apply to any redistributions of the source either in textual or binary form, with or without modification. As the project uses these libraries it will also be licensed under the modified BSD license.

Intellectual property of the dissertation is taken by the University of Brighton. The university requires access to intellectual property generated by students. As a condition of joining the university, students grant the university the right to use their work for academic purposes, including assessment and research

As the project does not make use of any human subjects for either evaluative or research purposes no ethical issues have arisen due to the work carried out in this project.

0.2 Requirements

After preliminary research, the following aims were laid out for the technical and qualitative aspects of the produced simulation, in regard to the behaviour of the simulation and the ants within it. Also listed are the learning outcomes which were to be achieved throughout the project.

Technical Aims

- To explore parallelism and implement parallel and sequential algorithms.
- To produce functional code that addresses the problem.
- To implement accurate collision detection.
- To implement a data structure that holds the information in the simulation.
- To develop custom algorithms suitable for this project.

Qualitative Aims

- To produce ants which have a sense of position and can navigate their environment
- To produce ants which appear to interact through the use of pheromones
- To produce environments of varying size in order to test the simulations scalability
- To produce clear concise and understandable code.

Learning Outcomes

- To be able to program functionally
- To have a better understanding of parallelism
- To have a good understanding of the programming language(s) that I use to implement my solution.
- To be able to analyse and critique algorithms effectively.

Technical Requirements Above all the aim of the project it to exploit the parallelism exposed by purely function in Haskell. The resulting artefact will be tested in three ways in order to determine whether it has made effective use of parallelism within the simulation. Firstly attempts will be made to produce functions which run the simulation in a serial manner in order to compare the runtimes of the parallel design with the serial. Secondly, varying loads will be placed on the simulation, this will be achieved by creating worlds dynamically from a set of parameters. Thirdly, the simulations design must allow for varying degrees of parallel granularity. The parameters introduced from these different levels of granularity can then be used to fine-tune the simulation to determine optimum settings and better assess where introducing parallelism makes improvements on run time and where it does not due to increased overheads.

Data Structures and Collision Detection The design produced should allow for collision detection to take place efficiently and accurately. Ants must not be displayed as intersecting at any point during the simulation. This may be achieved a priori through continuous collision detection as discussed during research or a posteriori by means of discrete collision detection with an adjustment phase to prevent intersected ants being shown. The data structure should also assist in pruning unnecessary collision tests from needing to be computed. This will ultimately increase the efficiency of the simulation and allow for a final product which scales well with the problem size.

Iterative Development It will be critical to test all code as it is produced, this not only saves time debugging but testing the final project build is more likely to be successful. The language Haskell, in which this project will be undertaken, allows this approach to development to be taken naturally through its rich and powerful type system. The use of type signatures as annotations to functions allows the programmer to specify the type of values the function should expect as parameters and the expected type of the return value this makes "it impossible to write code violating chosen constraints" [**?**]. As well as writing test cases and running the program to see if its functionality is correct, Haskell's QuickCheck package will be used to perform algebraic testing on key functions within the simulation. Given properties that functions must hold and knowing the data types of both the functions parameters and the returned result, QuickCheck automatically generates test data to feed the functions and tests

the result against the specified property to see if the function passes the test. Running QuickCheck notifies the programmer when properties aren't up held and allows for fast and rigorous testing of the application taking an algebraic approach.

Aesthetic Requirements The behaviour of the project's resulting artefact aims to portray a functioning ant colony. Ants should seek out food using exploratory strategies, communicate locations of food to other ants in the colony using pheromones and return to the colony's nest with food. The ant colonies increasing size should be reflected in the size of the nest, and the nests size in turn will dictate the amount of new ants generated by the colony. The nest's physical size will grow relative to the amount of food successfully brought back to the colony by the ants in the simulation.

Exploratory Strategies Ants will explore the simulation world by moving randomly away from the nest, each ant will have a direction property which when set allows them to track their location relative to where they have already been. One example of the use of storing directions within the ant is to detect when an ant has made a circle and returned to a location it has been before. Then an ant will then be able to make the decision not to take the same path as it had previously leading to more intelligent exploration strategies. Another example of an exploration strategy is moving away from ants which are not holding food and going in the opposite direction to ants that are carrying food. If ants carrying food are looking to return to the colony this would also be an effective strategy.

To communicating locations of food, once an ant locates food within the simulation, it will excrete pheromone which will be left on the ant and on the current location where the food has been located. Ants will also be tainted slightly with the pheromone from their current location, this coupled with the surface of the simulation dissipating pheromone every simulation step will allow ants to leave a dynamic path towards food sources by leaving a trail of pheromone. Once ants who are currently searching for food detect a pheromone indicating food they would be able to adjust their exploration strategy to detect which surrounding cell has the greatest pheromone level and use this trail to find food more efficiently.

The behaviour of returning to the colony, could also be achieved through the use of a different pheromone, or by allowing ants to retrace their steps. The preferred behaviour of the simulation's ants would use a mixture of the two, as ants are unlikely to remember the entire path back to their nest. Upon returning to the colony with food the ants will .The amount of ants added to the simulation will increase relative to how much food is brought back to the nest. Thus, the ants task of bringing food back to the nest enables the colony to fulfil its task of increasing in size.

listings language=Haskell

There is no standard way to model programs in Haskell and many different methods are used to carry an idea through from concept to code. Some programmers use modelling techniques similar to the Unified Modelling Language (UML), which is geared heavily towards object orientated languages with its use of class diagrams, commonly used in object-orientated languages such as Java. While other developers use concept maps and data flow diagrams to provide a general picture of the design before they begin programming. However, because Haskell is both a strongly and statically typed language it is a common practise to start constructing a program by working out the data types needed to complete the task. Then implement the functions for working with the newly created data types. Finally culminating in writing modules that bring together all the functions in a structured way [**?**]. The approach to modelling however, differs from project to project.

As identified during research data types can play an important role in program design, as Haskell is a statically typed languages the types of expressions will have to be validated by the type checker in order for the program to compile therefore it is practical to get the types organized within the design phase. UML is not limited to Object Orientated Programming. Even though it is founded in Object Orientated design it can be adapted a long way. The data structures are typically determined by the operations one wishes to run on them, so the usefulness of a diagram ultimately depends upon the kind of UML diagram being used and what is trying to be shown by the diagram. Package diagrams, Communication diagrams and Sequence diagrams may all be of use when designing an application such as a simulation in Haskell. Package diagrams can aid in making useful decisions as to what modules are necessary when importing, and also helping the programmer to make decisions as to what function should be exposed by each module. In approaches like the Actor model, discussed in the research chapter, communication between processes in necessary and mapping out such communications during the design phase would minimize bugs later on in development.

Ian Foster[**?**], identifies several key points to consider when designing a parallel algorithm these include, Granularity, Load Balancing, Dependency, Scheduling. Unless the problem already contains tasks which are by nature separate and independent in order to introduce parallelism to the problem it must go through a partitioning phase. The amount of individual parts the problem is broken up into is arguably the first thing that must be considered as this has a bearing on the following points. If the objective of the program is to produce a single result from the tasks evaluated in parallel there will have to be some communication between the process running the tasks in parallel in order to compute the final result. This results in an issue that the further the problem is split the greater the amount of communication that will occur to achieve the result.

When deciding on an approach to partition the problem it is important that the partitions are of comparable size as this will make it easier to keep the load balanced across the processors. If this is not the case, the efficiency of the parallel computation as a whole could always be increased by keeping all the processors available working on a task by introducing a pre-processing step to choose appropriate partitions. However, a pre-processing step may slow the computation as a whole as it may take more time to calculate the most effective partitioning strategy for optimal load balancing and run these partitioned tasks in parallel than it would take to process the problem with a less sophisticated strategy.

There is a danger that portions of a program intended to run in parallel may become concurrent if they are dependant on information other tasks. Dependancies should be sought out and where possible to identify how important a task depending on how many processes are dependent on the result it produces. These dependencies in within parallel program can be modelled using a Directed Acyclic Graph. If the amount of processes dependant on a task can be identified early enough the pogram can priorize task in order to create the opportunity for more parallelism. These decisions can be made before the program is compiled if enough is known about the behaviour of the application or the responsibility of which tasks to compute first can be given to a scheduler process.

The project's design began by looking at parallelism from a high level, as it is important to parallelize tasks a seperation on the data level was decided on, between ants, the simulations surface and pheromones. Operations that affected each of these datatypes could then run in parallel. Thus a divide and conquer approach was taken to the problem of the ant colony simulation. The design of the simulation was also an iterative process as the knowledge of both Haskell and parallelism grew throughout the early stages of the project the initial design concept evolved into a final design which better fitted the requirements and specifications set out in previous chapters.

The first design was list based and involved the free movement of ants with in the simulation (i.e. movements not being bound to a grid). The Ant type held two values of type Double representing its x and y coordinates in a two dimentional plane. Ants would be updated by a function which mapped across a list of Ant values. A seperate list would contain a list of pheremone drops which would be generated from the list of ants. But being stored seperately functions such as the disipation of pheremone could be handled in parallel by mapping a function which increased the radius of a pheremone drop locationand decreased its strength. Mapping a function like this at regular intervals in parallel with the ants movements would allow for a dynamic simulation with two tasks being able to be carried out in parallel. However this idea was deprecated in favour of an approach which offered further parallelism.

The next concept for the data representation of the simulation world was to produce a list for every type of item in the world. For instance there would be an Ant List, a Food List and a Surface type List. Each item in the list would also be given a location to represent where it was in the simulation world. Each list could then be processed in parallel. Copies of each list would be made for lists which needed other lists in order to compute their changes. Once each list had updated the old lists would be discarded and copies of the new lists would once again be made for the next simulation step. Each item in every list would be paired with a location value of type Int Int. This combined with a limitation that every element within the simulation is the same size, allows for fast checks to be performed as to whether an ant is within range of a certain pheremone, or if an ant is at a particular food source. These lists of data tagged with locations could also easily be split allow for partitioning of data by grouping data with location tags within a certain range together, processing each of these groups in parallel. This design was dropped due to the preprocessing step needed to split the data up for parallel processing.

In order to improve the design an effort was made to select a datastructure which held more information about the simulation. Using a Graph representation of the world collitions are cheap and the possibilty for ants intersecting is non existant. The adjacency list of the graph always holds the locations of the surrounding nodes and making inquires as to what is in surrounding cells is straight forward as only the nodes in the adjacency list have to be queried. An ant can therefore make a decision on which node it wants to move to in the graph from its current node, and a simple check to see if that node is currently occupied before it moves is computed. If the node is full the ants request is not carried out and no movement occurs.

This design maintained the attribute of confining the ants movements to set locations, however now these locations were represented by nodes within a particular graph. When looking at previous large scale simulations which tried to exploit the benefits of parallel programming it was noted especially in the simulation produced by Intel that identifying tasks which can be computed independantly of other tasks and arranging the computation of these tasks is key to achieving speed up. In previous designs a seperation has been made between pheromone and ant data to allow for operations which run in an embarassing manner. This was maintained throughout this design using multiple graphs to represent pheromones and food data seperately from ant data.

The final design that was settled on was graph based, using graphs to represent individual portions of the world, referred to as quadrants, that could be processed in parallel. These graphs were each held inside an overarching world graph which was used to derive the positions of each of the quadrants in the simulation world. Again, graph representations of the world were then responsible for holding certain information. The concept of a Nest graph was also intro-

duced, which held the location and would monitor the expansion and possible reduction of the nest's size depending on the success of the colony. Further, seperate, graphs holding pheromone information and food locations and quantities would also be passed around in the simulation as functions required their infomation for processing but their processing for the next simulation step could be carried out in parallel as they are seperate and independant data structures.

This splitting of the data representing a world into quadrants introduced a new problem, the movement of ants between quadrants. Divide and conquer can only work when all the sub problems are independent. In a purely functional setting, elements of a data structure being computed by map cannot see the effects of the computations on other elements. If the order of application of a function to elements in the datastructure is commutative, for example the movement of ants elements in multiple quadrants, it is possible to reorder or parallelize execution. If an ant is moved in quadrant one then an ant is moved in quadrant two the resulting world would be the same if the operation were performed on quadrant 2 first, or in parallel. This is the opportunity that divide and conquer algorithms and MapReduce frameworks exploit.

One proposition that was debated as a potential solution to the problem of ants moving between quadrants, was a message passing system based on the actor model. Each quadrant would be processed by a seperate actor on a different thread, processor or machine and when ants needed to move between. Their details could be sent to the thread processing the quadrant the ant wished to move to as a request. The request could be then accepted or a message could be sent back with the ants details if the request to move the ant was declined due to the node the ant wanted to move to being occupied. Another approach that was considered to solve the problem of passing ants between quadrants for processing was the use of software transactional memory. All methods updating edge nodes would become atomic blocks allowing for updates to nodes on the edge of a quadrant to be recorded as transactions and when the data representing the edge needs to be shown there would be the assurance that every operation on edge nodes had seen a valid state of memory due to the way atomic blocks function, meaning that no ants would be over written by other processes running concurrently.

The finalsolution to the problem of moving ants between nodes was to process ants on the edge seperately to other ants in a quadrant. By taking pairs of quadrants and running functions which handle necessary transfers between the two quadrants before processing the ants which are not on the edge would ensure the parallel phase of processing would not affect other quadrants. Another issue arises when handling the edges of quadrants and the centers in different steps. Ants moving across an edge are processed more than once by both the edge ant movement function and the function that handles ant movement for the quadrant as a whole. To prevent ants from appearing to jump or increase

speed when passoing between quadrants in the simulation world, a list of ants processed for each quadrant should be maintained in order to prevent these ants being processed again during the parallel phase of ant movement.

The initialisation of the simulation would begin by taking an initial set of parameters. These parameters would define the size of the simulation world in terms of the World graph, how many quadrants each world would hold and the quadrant size the square root of how many nodes each quadrant would contain. The initial number of ants the simulation would begin simulating would be a further parameter along with the amount of food within the simulation on start up. The program would then initialize each these graphs as empty and randomly determine the position of the nest within the simulation world by updating the nest graph. Following this the ant graph would be updated by adding the amount of ants specified by the simulation parameters to the ants nest in random positions in proximity to the nest. The next step would then update the food graph with the amount of food locations required by the parameters, these are to be added to random locations within the simulation world with the exception of the nest location. The program then moves on to the simulation loop.

In order to produce more parallelism within the simulation loop the dependencies of important functions within the simulation were analysed. In order to determine which functions could be run in parallel while maintaining consistant data and which functions relied on the results of other functions and therefore had to be computed sequentially. Generation functions which initialze the simulation have been identified as follows:

- The generation of empty Nest, Pheremone, Ant, Food and Nest structures of uniform size.
- The generation of a Nest location.
- The generation of the simulations inital ants.
- The generation of Food sources

The following function which the simulation loop composes of are as follows:

- Move ants
- Add new ants to the simulation (Birth ants)
- Remove ants from the simulation (Kill ants)
- Removing food from a food location and allowing an ant to carry it (Harvest Food)
- Removing food from an ant and placing it in the nest (Deposit Food)
- Transferring pheremone from the simulation surface to an ant.

- The evaporation and dissipation of Pheromone on the simulation surface (Dissipate Pheremone)
- The transfer of ant pheremone to the simulation surface
- Adjusting the size of the nest.
- Adding new food sources or increasing old food sources (Update Food)

Each of these proposed functions were then analysed to determine what datastructures they would update and produce as results. A diagram was then produced showing each of the structures linking them to the functions which would update them [Fig], along with a diagram showing the information which each function will need in order to compute its result [Fig].

Through the use of these graphs, due to the indication of the types returned and the purity of each function, dependencies could easily be higlighted within the data being processed. This allowed for a decision to be made on in what sequence the functions should be called. For instance, the function to dissipate pheromones throughout the simulation surface would not be able to run in parallel with a function which updates the updates the pheromone levels on the surface of the simulation with respect to the ants pheromone levels and positions. As both functions produce a new instance of the pheromone graph. Within the main loop six steps take place are as follows.

- The current state of the pheremone, food, and nest worlds are passed to the ant processing functions in order for the ant world to update according to them.
- The following steps can take place in parallel.
  - The pheremone world is updated with respect to its dispertion and evaopration functions.
  - The food world is updated with respect to its generating further food in current sources and new sources.
  - The more sequential process of updating the edges of each quadrant and stitching them together, allowing the crossover of ants.
- The internal processing of each ant quandrant with respect to ant movement is carried out in parallel.
  - The nest world is updated with respect to its changing size
  - The ant and food quadrants update with respect to ants harvesting food where possible.
  - The pheremone quadrants update with respect to ants dropping off pheremone trail.

- The ant quadrants update with respect to ants absorbing the pheremone from the surface.
- Simulation step is complete and viewable, calling itself recursivley to continue looping.

0.3 Planning

0.4 Software Development Model

This project will take an iterative and modular approach to development. This approach can also be seen as "a way to manage the complexity and risks of large-scale development"[**?**]. It is a useful coding practise to develop small modules which are capable of functioning independantly from the code as these modules can then be reused, in developing other programs. There is also another advantage to be drawn from modular development, each module would be able to be tested sperately which would in turn ensure less bugs when assembling the final product. Another advantage of the iterative incremental approach is that the developer is able to take advantage of what is learned during the development of earlier, increments of the system. As learning comes from both the development and use of the system [**?**].

0.5 Important Tasks

The following is a break down of the project into smaller iterations this can then be used alongside the various project deadlines the project entails, to produce a Gantt chart of how the project's workflow will be carried out.

- Representing an ant
- Representing the worlds
- Individual ant behaviour
- Implement dissapating pheromone levels
- Implement the critical simulation functions
- Parallelize algorithms further
- Extending the solution through application to a distibuted system.

0.6 A Schedule of Activities

The produced Gantt chart attached to this report visualizes the following information. Throughout the following paragraphs a description of the plan for the

project will be detailed in the form of a schedule. Although this plan looks at the project lifecycle as a fixed stages to be completed in an order, work further work on refactoring tasks which have been achieved earlier on in the project will be an going process throughout the later stages of the schedule.

October-November Throughout the course of October and November the project will focus mainly on gathering resources and research. Much of the research will be conducted in the functional language Haskell and small experiments will be made to demonstrate any concepts that have been learnt. Further research will be carried out in areas of the project that will pose potential problems that need to be dealt with later on in the implementation stage of the project. Even at this point in the project the implementation process will begin. As development will be iterative small aims will be set such as representing a single ant and then these aims will be expanded upon.

December-Janurary In the months of December and Janurary the code produced in the first section will be expanded upon to produce a working base system. At the end of this this period a simulation of the ant colony should be able to be run. The simulation should include the all the basic functionality of the final system. Ants will be able to move and collide. Each ant will respond to its surroundings by referencing its own basic behaviour tree.

February-March In Feburary and March the focus will be on improving the algorithms which have been used within the system. Making them more efficient and looking at structuring more of them to work in parallel. At the end of this period the project's delieverable, the simulation, will be in a state that is presentable with as few bugs as possible. Towards the end of this period if things are on schedule the project's focus will shift from improving the base simulation to adding extentions and making it more usable.

April During the final few weeks of the project there will be the fixing of any bugs in the simulation and finishing up any extentions that may be added to the project. During this time the projects focus will be on documentation and producing an informative and evaluative technical report.

0.7 Risk Analysis

Throughout this project there are several things that could go wrong, such as loss of work, hardware failure and the inability to surpass certain bugs which may be encountered throughout the project. To minimize the chances of loss of work source control tools will be utilized in the form of muliple git repositories. This will not only backup the project but keep a track of changes and monitor its evolution. In the event of hardware failiure all the software required for the project is freely available and it would be possible to set up for work either in

the University or on a replacement system with minimal effort. There is also a chance that bugs may be encountered which appear impossible to overcome this is a risk that can only be managed through planning and research. Should difficulties be encountered there is a vast range of resources for functional programming online and a Brighton functional programming user group, this would provide a great opportunity to increase my knowledge but should it not be possible to find a solution the only course of action would be to alter the future plans of the project.

listings

0.8 Implementation

This chapter will discuss what has been done so far in terms of implementation and the problems that appeared during the process. The project emerged from the explorations into the programming language Haskell which began during the research stages of the project. Progress was initially made by laying out the list structures for representing the simulation world, the first of the designs elaborated in the design section. During this phase concepts like pattern matching and recursion were introduced which became essential later on in the project. A module was created holding all the data for an ant so functions could be called from it in order to trigger an ant's behaviour. At this point, the project had the ability to map a function across the list of Ants, to move each one. In a functional language mapping is the processes of applying a function to every element of a list, map, itself, is a function which takes both a function and a list and carries out this operation. Here came the first major problem, the ant module contained four functions for Ant movement, moveNorth, moveEast, ect. and mapping one of these functions across the list of Ants would cause all of the ants to move in that particular direction. There were two solutions to counter this problem; firstly, a general move function could have been written which took a random value and used it to make a selection between the four subordinate functions which moved an Ant in a specific direction.

[small diagram]

Secondly to apply a function which split into lists a list of four lists, with each element within that list containing ants which all wanted to move in the same direction. In this case the four directional movement functions could each map over a smaller subset of the list which where all of which wanted to move in the same direction.

[small diagram]

The former solution was pushed back due to a lack of understanding Monads, the Random Monad would have been required to develop this solution making use of the Prelude module, System.Random (The Prelude is the name given to Haskell's standard library). So things pressed on with the second approach to the problem. While writing this decision making function to split the list of Ants

into a list of lists it became clear that this was not the best approach to take as parallelism would be limited and especially calculating the concentration of a pheromone at a specific location would require a lot of processing which would introduce a lot of dependence. Use of a non-grid based system was the main issue when looking at the complexity of calculating pheromone concentration at a specific location. This stems from the fact that an instance of pheromone may have a location which does not necessarily correspond exactly to an ant's location but is close enough to affect it. An algorithm to calculate the exact concentration of pheromone at a specific location would need to check every pheromone instance in the pheromone list and to see if the location of the ant falls within the radius of the pheromones effective area. This would involve excessive and unnecessary checks which would fail as every Ant would have to be checked against every instance of element in the list of pheromones.

The checks would assume a circular bounding volume approximates the ant, to a check if a proposed ant move result in a collision the formula below is taken.

sqrt(sqr(m) + sqr(n)) (2)Where m is the difference between the two x locations and n the difference between the two y locations. This would be then be compared to the sum of the two radii, be it ant ant or ant phereomone, if the sum of the radii is less than the result of the forumla above then the move cannot take place.

To detect if an ant can access a food source a a specific location the equation of a circle is taken.

$x^2 + y^2 = r^2$(3)

To determine a food location (p) lies within the bounding volume of an ant where $p = (a, b)$ then if $a - x^2 + b - y^2 <= r^2$ where x and y is the location of the ant and r is its size as a radius, the food source is touching the ant or under the ant and therefore can be harvested.

The problem of limiting all these pair wise checks between every ant pheremone instance and food location was not the focus of this project, however would make a great consideration for future work as this would lead to a more liberated and naturalistic simulation. To resolve this problem the decision was taken to limit movement of ants and locations of pheromone instances to a grid based structure, this provided a way for checks to be performed on the pheromone list in order to determine whether or not a certain amount of pheromone was at specific location. In order to reduce the amount of checks multiple lists were employed to represent each row within a grid for both ants and pheromones. With every row containing the number of elements as columns within the world there was a requirement to store the absence or presence of data at a specific location within the grid. This was done through the use of Haskell's parametric

data type Maybe which can either hold a Just value or Nothing. This enables functions to be written which would accept the absence of an ant as a valid type as well as valid ant information so long as they were wrapped in the algebraic type Maybe. Thus, abilities such as mapping a function over the data representation of the simulation world could be preserved.

Until this point in the project Ants had been stored as a parametric data type which accepted (The parameters for an ant and their types) in order to retrieve values from them, functions needed to be programmed to retrieve specific values from data of an ant type. To retrieve values from a parametric data type Haskell's process of pattern matching is employed by using it to bind the values of the parameterised type to variables and then by providing the variable as the result of the function it is possible to retrieve the desired value from the data type. This is also known as deconstruction [**?**].

data Ant = Ant Location Direction deriving(Show)

moveAnt :: Ant -¿ Ant moveAnt (Ant (Location x y) d) = Ant l d where l = updatePosition (Location x y) d

Here the two values that make up an ant are broken up between the variables x and y, for the location values and d or the direction. These values are then passed on to another function which produces a new value of type Ant making use of the values that have been pattern matched out of the original ant.

This would soon become cumbersome as the parameters of the ant data type would change to conform to changes within the design. As a result the Ant data type was transformed a record data structure, in Haskell this gives the advantage of providing functions which extract data from each field of the record. This resulted in the code being cleaner and easier to read. With the arrival of a data structure which held values for all the possible locations within the simulation world it was no longer necessary for ants to hold the value of their current location as the simulation world's data structure could be queried in order to acquire the values in the surrounding cells. By removing this redundant data not only was the ant data type simplified but also the information held within the data type representing a pheromone could be reduced to a base type.

At this point in the project it became increasingly important to retrieve the surrounding elements of a cell in order to provide an ant the information with which to make its next move. This involved the generation of a few utility functions. These functions when provided the size of the grid world's width or height and the cell of interest made use of the modulus in order to deduce the surrounding cells. From this position it was noticeable that these functions would be called heavily just to access information that would never change. A data structure which encapsulated the information of the connecting cells was needed. The solution to this came in the form of a graph representation of the simulation world.

A graph can be represented in memory as an adjacency list, where each node is coupled with a list of the nodes it is adjacent to. In order to represent a graph in Haskell the Data.Graph library was used which was produced by The University of Glasgow and is licensed in a manner similar to the BSD License. It provides functions for the creation of custom Graph types when provided an adjacency list and a list of vertices. It also provides a few utility functions such as indegree and outdegree which provide the edges going into and out of each node. Each node in the graph is a type variable allowing graphs to be created holding anything from a simple Double representing a concentration of pheromone to a complex record representing an ant, as long as all the node in each graph are of the same type. The vertices of the graph are Integer labels given to each node within the graph in order for functions to be written that can access a node given its vertex in the graph. Throughout the course of early stages development functions were written two automatically generate graphs of certain sizes when given a list of elements to be stored in the graph nodes. These functions grew to become the GraphOps.hs module which can be viewed as a wrapper for the Data.Graph library.

The first functions programmed to manipulate these graphs dealt with ants movement, this was a key feature and one of the first to be developed. As an ant could only move to an empty node, a swapping function was written which when given the location of two nodes within an ant graph would swap their values. To do this a function to return the node of a graph at a particular vertex was mapped over the graph's list of vertices. This generated a list of nodes. A function was then applied to the list that given two vertices in the list of nodes a list would be returned with the two specified nodes swapped. A function which built a graph from a list of nodes was then executed to produce the final resulting ant graph. Since data in Haskell is persistent this resulting graph couldn't be assigned to some storage. Rather the result had to be used as soon as the function had been called, this took time to adjust to.

Although the state of the project would now allow operations which provide an update to one of the simulation world's data structures to run in parallel with an operation providing an update to a different structure; the project could still achieve greater amounts of parallel speed up. Currently to produce an updated graph representation where all the ants in the graph have been processed, giving them the opportunity to move in their desired direction, if possible. Requiring all the ants to wait for their turn to be processed in exactly the same manner as ants whose actions have no affect on them is another remote area of the grid is a scenario which adapts well to the divide and conquer method of parallelism discussed in the Context and Background Research section. In order to take this approach it necessary to be able to split the computation up into sections. To do this the same graph which represented an Ant world was now renamed a Quadrant, so that several quadrants could comprise a world, as seen in Figure X in the design section. These quadrants were then stored in nodes of a graph

this would allow for simple tracking of the edge relationships of each quadrant.

With the partitioning of the simulation world into quadrants now in place a function, for example the movement function, which updated a quadrant could be applied to each quadrant there by achieving parallelism within the simulation's world graphs. When working with the world graph it was necessary to keep in mind that what was stored in each node was actually a tuple containing the graph and two additional functions which performed operations on the graph. For this reason the world graph's contents could not be printed to the terminal directly using my existing function to print graph vertices, a function to print a quadrant from a world now has to extract the quadrant from the world before printing it.

The first approach taken to generate parallelism within the ant world graph was to use the 'par' and 'pseq' functions from the Control.Parallel library. These functions are one of the earlier approaches to achieveing parallelism within pure code. The expression (x 'par' y) would spark the evaluation of x and returns y. Sparks are not executed straightaway but are queued. In order to spread the available parallelism over the CPUs, if at runtime it is detected that there is an idle CPU, then a spark is converted into a thread from the list of queued sparks, and is run on the idle CPU. When attempting to use these annotations it was difficult to compile the program. Using the par function a elements in a list can be mapped over take two, three or even four elements, with compiler knowing that the evaluations may be carried out in parallel. Below is a modified version of map which pattern matches out elements of a list and applies a function which is passed as a parameter to elements of the list using the par function.

map' :: (a -¿ b) -¿ [a] -¿ [b] map' $_{[}$] = []$map'function(w : []) = [functionw]map'function(w : x : []) = fx$'$par$'$[functionw, fx]wherefx = functionxmap'function(w : x : y : []) = fx$'$par$'$fy$'$par$'$[functionw, fx, fy]wherefx = functionxfy = functionxmap'function(w : x : y : z : zs) = fx$'$par$'$fy$'$par$'$fz$'$par$'$(functionw : fx : fy : fz : (map'functionzs))wherefx = functionxfy = functionyfz = functionz$

Such a function will allow for up to four elements to be processed in parallel but in order to produce a clearer, more generic and efficient specification of parallel evaluation strategies from the Control.Parallel.Strategies were utilised [**?**]. This was mentioned as one of the methods of achieving parallelism in Haskell during the research section and was the next step taken in attempting to achieve parallelism within the ant world graph. Evaluation Strategies allow the programmer to introduce parallelism in pure, deterministic Haskell programs and allows them to remain so. Parallel strategies provide various compositional strategies which generalise the par and pseq annotations to a higher level. Below is a code snippet of one of these strategies being used to process a list in parallel, the amount of processors utilized is limited by the machine.

runSimParallel :: [(GraphATuple, GraphPTuple)] -¿ [Int] -¿ [GraphATuple] runSimParallel zippedQuads noProcs = parMap rpar (processAQuadrant noProcs) zippedQuads

Now that the simulation world was divided up into quadrants, provisions had to be made to allow ants to move between them. Several approaches to this problem were looked at and have been previously discussed in the design section including use of software transactional memory and using a message passing system influenced by the actor model. These approaches were dropped because of difficulty using the libraries which offered the functionality needed to take the project in this direction. STM makes use of the Control.Concurrent.STM library and The only design which was implemented was the table of no processing lists. This began by developing functions to extract all the nodes along the edge of a quadrant when given the size of the quadrant and the direction in which the edge lies. When applying this to both a pheromone quadrant and an ant quadrant provided two lists nodes from the quadrant edge. The same could then be done to another quadrant edge to produce a similar pair of lists for an edge opposite to the previous edge.

These four lists of nodes were then zipped together with the vertices on the graph to which they correspond. The zip function in haskell takes two lists and produces a list of pairs where each pair is an element from each list at a particular index. Each graph's lists were then coupled to produce a final edge pair, this edge pair could then be mapped over and would contain the information of what was at a particular vertex what was on at the vertex on the opposite quadrant, and other useful information to aid in transfer of ants between quadrants. These edge pairs alone would not be sufficient calculate the new state of the quadrants after the edge ants had been processed. The current ant graphs and pheromone graphs were paired as well as a pair holding two Direction types. This served to indicate the relationship between each of the pairings being made.

This was all stored within a datatype called StitchableQuads which allowed this group of data to be passed around easily from function to function. Prior iterations saw functions with an increasingly longer list of arguments, which made code difficult to follow. Encapsulating these different values within a data type remedied this problem. Now the data that needed to be manipulated was all together and easily reachable within the StitchableQuads type the next step was the processing of this value. The ant Edge pairs would be processed sequentially by running a function over its lists which processed the ants in a similar way to the ants within a quadrant however allowing them to sense outside of the quadrant. With each pair of opposing edge nodes there were three scenarios that could occur and each needed to be handled in a separate way.

Neither node holds ants One node holds an ant Both nodes holds ants

This is modelled implemented in Haskell code with the function below. antScenarios qs pos currAnt adjAnt — (isNothing currAnt) (isNothing adjAnt) = procEdgeAntAtNode (1+pos) qs — (isJust currAnt) (isNothing adjAnt) = loneEdgeAnt qs True pos — (isNothing currAnt) (isJust adjAnt) =

loneEdgeAnt qs False pos — (isJust currAnt) (isJust adjAnt) = doubleEdgeAnt qs False pos

If the opposing edge nodes both held no ant the function called itself recursively with the next pair of opposing nodes. Should one node hold an ant while the opposite edge node held no ant, the ant is processed updating updating either one qudarant if it chooses to move back into the quadrant or both quadrants if it moves out of it's quadrant. The no processing list for each quadrant is updated as necessary to ensure the ant isn't processed again during the general movement of ants quadrants. Finally if the scenario occurs such that two ants are present checks are carried out to see if either ant wishes to move back into their quadrant. If an ant wishes to move back this is processed before ants can move out as this will free space to allow the other ant to move across the quadrants.

In order to produce the list of nodes not to process throughout general movement of ants within quadrants whenever an ant was moved by the 'stitching' algorithm it was added to one of two lists held within the StitchableQuads data structure, one list for each quadrant represented. Each of these lists is then appended to a list of vetices which have already been processed for each quadrant. As quadrants have up to four edges which can be processed these lists grow as the edge node processing step completes. Finally, these lists of vertices are passed back into the function which processes the complete ant quadrants so that the vertices that have already been processed can be avoided.

At this point it was necessary to begin printing ant quadrants visually in order to test if the functions were producing the desired behaviour. First a function was produced which allowed for the viewing of specific quadrants using haskell's guards conditionals where placed on the functions parameters and if ant was at the given location an 'O' was printed while if no ant was present an 'X' was printed. Making use of the modulus function in conjunction with the quadrants size allowed for each row of the quadrant to be printed on a seperate line. Printing quadrants in this manner forced the values produced by functions which operated on them to be fully evaluted. This drew attention to bugs not only in misplaced ants as a result of incorrect algorithms but also some functions, such as procEdgeAntNode which called itself recursively through the list of edge nodes in a StitchableQuads datastructure, hung indeffinately.

In the process of tackling this problem it was discovered that debugging Haskell code is also very different from solving bug in imperative code. A runtime debugger that tracks a variables value is almost redunant as variables can hold 'thunks', unevaluated values, due to Haskell's property of non-strictness. It was discovered that a simple way to evade this problem is to break large functions down into smaller ones and monitor the return values of each of these functions closely using ghci, the interactive Haskell console. However, in this

instance this approach only allowed me to narrow down the problem of the infinite loop to the function procEdgeAntNode. Further research into the issue introduced a program for code coverage called HPC (Haskell Program coverage tool). The program highlights the functions called during the running of a function and aided in confirming that the situation was truely an infinte loop. It was not until the decision was made to enable all warnings on compilation with the command line option -Wall being passed to the compiler that the vast amount of potential program problems were brought to light.

Most of the warnings were produced by non-exhaustive pattern matches and missing type signatures which had to be inferred by the compiler. After working through and resolving warnings which presented opportunities for program crashes shadowing warnings were noted. Shadowing occurs within do notation when a variable is created within a function while another variable of the same name already exists in scope. This problem created a misunderstanding and so while it was assume a parameter was being incremented every time the function was run, it was being passed unchanged. Other warnings which were reduced were the defaulting of type constraints. In some places by using more specific type signatures allowed for less errors to occur but sometimes this was necessary to be non-specific about types for polymorphic functions.

As discussed during research the amount of speed-up achieved by a parallel program is limited by it's sequential portions. In order to increase the the speed-up the parallelism was extracted from the sequential process of processing the edges of quadrants. The list of pairs of quadrants which shared edges was split into a list of lists where each list contained a quadrant pairs such that no quadrant appeared twice. This allowed for each of the lists to be processed in parallel as no quadrant would be processed by more than one seperate process. By introducing parallelism at this stage the sequential amount of the program, the 1-P term in Amdahl's formula, the limit on the minimum speed the program can run at is reduced. This results in greater parallel speed-up.

Lazy evaluation is utilized within the functions which manage the transfer of ants between quadrants, in the module QuadStitching.hs. The function holds a pair, each element of the pair in turn holding a pair of calculations. Each calculation updates one of the graphs within the StitchableQuads instance. The following code snippet was taken from the swapIn function (found in the QuadStitching.hs module) and moves and changes the the direction of an ant at a node in one of two graphs stored in the StitchableQuads data type. A graph selection function (gs) is applied to the pair of calculations so only one side is ever used. Due to Haskell's non-strictness this also means that only one side is ever evaluated and the function remains efficient, not evaluating unnecessary functions.

let di = gs (((setDir (fst$antGraphsqs$)nd1d), (snd$antGraphs qs)), ((fst$antGraphsqs$), ($setDir$(snd$antGraphs qs) nd1 d)))

When writing functions to print a world graph to the console for testing, the function to print quadrants graphs could no longer be used. Instead of printing a world's graph one quadrant at a time one row from each quadrant had to be printed for each row of quadrants in the world. This was done through the use of the modulus function in conjunction with the size of both the world graph and the quandrant graphs. This process involved a lot of debugging by breaking functions down into smaller more specific functions and ensuring each produced the desired output by supplying test parameters. The functions which produce these easier to view worlds can be found in the ConsoleView.hs module and are called prettyAntWorld and prettyPherWorld, for viewing the ant and pheromone world graphs respectively.

Manipulating the pheremone world graph was easier in comparison to ant movement disappation of pheromones was as simple as mapping a substraction function of a constant value over the nodes of each pheromone graph quadrant. The spread of pheromone was similar again mapping over each node increasing the values of the nodes in it's adjacency list by a constant fraction. To increase the simulations complexity further the introduction of ant behaviour followed. Until this point ants would base their decision as to which direction to move next soley on which of the surrounding pheromone nodes contained the highest concentration. An attempt was made to alter this behaviour by introducing randomness to the decision. As Haskell is purely functional any random values generated are monadic types, such types can not be used freely in pure code. In order to break free of the constraint the function to generate the random numbers had to be produced in a function which produced a side effect. The generator functions are in the RandomNums.hs module.

In order to make use of the monadic values the values inside where bound to a function using $< -$ so that they could be passed as parameters to pure functions. This approach worked as long as functions making using these bound monadic values where in the function main, as main's return type is an action, main::IO. Below is a code snippet showing how these random values were retrieved.

main :: IO () main = do rnumbers ¡- genRandoms 1 20 testRun rnumbers

Unfortunately when attempting to pass these random values through to the decision making functions a bug was generated and the program would no longer compile. The problem was caused by the pMap function only taking two parameters, this problem was solved when the list of nodes that had already been processed by the edge processing function had to be passed into the function by means of partial application, but when partially applying the extra string of random numbers for ant behaviour errors were thrown, which have not been successfully debugged prior to the project deadline.

In order to initialise the simulation as designed a starting amount of ants had to be added randomly to the simulation. To do this two infinite lists of random numbers were created and zipped together to provide an infinite list of

random coordinates. Then functions were written for the nest, ant and food quadrants which took these coordinates in as a reference a node amongst all the quadrants. This task again involved the use of the modulus function. This set of functions calculated the quadrant these coordinates fall in and then the precise node within that quadrant. The functions cause execution to terminate should they recieve an invalid coordinate which does not map onto the world with the use of the error function which has the type is String-¿a. Using the error function with different messages in the different functions to makew it possible to locate problems. However, sometimes when these functions run, due to randomness, they throw an index out of range exception. Even with checking the values of each coordinate and throwing a custom errors on invalid coordinates the problem still has not been located.

Difficulties where encountered on first use of the Data.Graph library. As this was the first time being exposed to real functional code outside the protection of tutorials it took time to grasp the concepts behind how the Graph API worked and how it was used. Looking back, it may have chosen a different API to implement my graph Data structure as Data.Graph isn't a functor there for it doesn't export a function to allow the user to map over it, this would have produced tidier code. But, with functions produced such as processAntsInGraph found in the Quadrant.hs module the functionality of using a data structure, which was an instance of the Haskell type class functor, could be reproduced.

Type ambiguity was often the reason the code wouldn't compile. Sometimes when programming helper functions, because what is passed in is already of a known type, it is easy to start programming a function without laying out an explicit type signature. Haskell inferrs the types of functions correctly however if the wrong assumption is made about a type passed to a function many type errors are thrown when functions start to be applied and composed. These errors occur at the location the of the functions application, in order to find the root of the problem type signatures must be applied to the offending function and an error will occur when Haskell's type checker runs during compilation.

0.8.1 Testing

As mentioned during the research section, there is a haskell library called QuickCheck which provides the ability to run algebraic tests on functions produced in Haskell. The Quickcheck properties were generated for the GraphOps.hs module, initially there were issues with the test running but never reaching completion. It was soon realized that the values quickCheck generates for it's tests needed to be restricted in order to supply the functions with realistic valid data. For instance, checking a generated list's length was equal to the given parameter squared, with larger intergers ends up with this the test calculating the length of extremely lengthy strings causing the test to hang. Also, the a function might only work on positive integers, quickChecks automatic generators which pro-

vide random data given the type do not offer this flexibility. Therefore, custom generators had to be programmed.

instance Arbitrary IntSmall where arbitrary = do i ¡- choose(0,1000) return $IntSmalli$

The code above produces a value of type IntSmall which is can be a value between 0 and 1000 this provides the functions within GraphOps realistic values to be tested against. A similar generator was also constucted to produce tailored lists called NodeList in order to test the functions used to organise data for dynamic graph generation of variable size.

The project was tested at various stages once parallelism had been introduced by Threadscope, a haskell profiling tool. Threadscope allowsthe parallel performance of Haskell programs to be analysed. Using Threadscope it is possible to check that work is well balanced across the available processors and spot performance issues in regard to poor load balancing and garbage collection. Below is a view of parallelism that was achieved earlier on in the development process, the first time that both cores of the processor were in use simultaneosly during the execution of the simulation.

The figure below shows parallelism being achieved in the current build of the program. This shows that the mapping of the tasks between the two processors is too small to generate parallelism for a lengthy amount of time. This is due to either the problem sets being assigned to each processor being two small and not computationally expensive enough or because each in each simulation step spent more time in the sequential task of collating the information from the graph for printing.

Further Work and Conclusion

There many areas in the project that need to be looked at,

hangs over 12 by 12 Currently the simulation has a fixed partitioning system, future work would aim to make this dynamic to ensure better load balancing through the use of Vonoroi, Barnes - Hut, algorithms

During the final weeks of development the visualization of the simulation was revisited in order to produce a more user friendly and easier to view simulation.

The project could be also expanded to make use of more than one processor by way of distributed computing. Extension the actor model could be implemented in a distributed fashion.

Aspirations of the projects artifact where very high throughout the initial phase of the project and the undertaking of some of these plans in a new language was possibly an over estimation of what could be achieved within the time frame the project allowed. However, as the realities of what was being undertaken increased through research and increased understanding throughout the implementation phase these expectations were adjusted. When taking this into consideration, the outcomes of the project... There have been many lessons learned over the course of this dissertation project.

More progress with a language like scala. Printing the world Quadrant seperatation variable world size Monad use of IO Use of Random

The future of computing depends on parallelism (for efficiency), distribution (for scale), and verification (for quality). Only functional languages support all three naturally and conveniently; the old ones just dont cut it.

My expertise in the the area of functional languages, parallelism, and xxxx list xxx have been significantly enhanced (reasons).

Incomplete however the project has demonstrated an algorithm with potential to expand to Using libraries like debug trace was ineffective