Context and Background Research

0.1 Research overview

In this project the problem of simulating the independant 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 will have a realistic limited field of sense. With each ant making decions based on their their own tasks, independantly; 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 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.

However it is important to mention here that the realisitic 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 programatically. Firstly looking at various programming paradigms before looking at different simulations. Then research will continue further on methods mentioned in this disscussion that seem most likely to provide a solution.

0.2 Ant Colony Generalizations

Listed here are the aforementioned generalizations or preconceptions about ants and their colonies, that will be assumed as factual for the purposes of this project. All ants need both food and water to survive, according to the Center 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 [**?**].

0.3 Problem Area

When the concept of this simulation is broken down it is evident that a crutial 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 loooking at first representing small amounts of ants, then looking at getting collision detection working 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 doesn't drop to a point where it is no longer accurate. This is important to note as when we look for a programming approach in the research stage of the paper it would be sensible choose an approach that inherently produces solutions capable of scaling well as the problem size increases so that performance doesn't dip when the demand on the system is increased.

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 attributted 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.

Chapter 1

Research

1.1 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 aim to address. 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, taking this further to look at why these paradigms are more efficient at dealing with such problems. Getting more specific, the focus will then be turned to what programming languages are used for this type of problem, again looking at why certain languages are perferred. 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.
- Analyze other large scale simulations.
- Look at approaches to large scale collision detection.
- Consider possible ways to represent the information in the simulation visually.

1.2 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 breifly at these four paradigms and go on to mention a few others which might be of interest considering the problem.

1.2.1 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 operate by performing one operation at a time, on a certain pieces of data retrieved from memory, in sequential order. According to Backus [**?**] the man who cointed 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. The storage is then is manipulated by the program in just as the variables and assignment statements with in the program dictate. Imperative programming languages provide a variety of commands to provide structure to code and to manipulate the store.[**?**]

Each imperative programming language defines a particular view of hardware.

These views are so distinct that it is common to speak of a Pascal machine, C machine or a Java machine. A compiler implements the virtual machine defined by the programming language in the language supported by the actual hardware and operating system.

As far as we were aware, 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 [In R. L. Wexelblat, History of Programming Languages, Academic Press, 1981, page 30.]

1.2.2 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 functionas 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 be tested seperately too allowing the ability for tests to be focused on modules can result in a greater amount of more specific tests which are more likely to catch errors in code.

"It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications ... use of threads can be very deceptive ... in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. 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 ..."

1.2.3 The Functional Paradigm

Object orientated programming has been described as "the antithesis of functiontional 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 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 it can even be constructed by composing functions. This offers a stronger "glue" to combine the modules of your program. A function that evaluates some expression can take part of the computation as an argument for instance, thus making the functio modular to a futher extent. You could also have a function construct another function. For instance, you could define a function "differentiate" that will differentiate a given function numerically. So if you then have a function "f" you could define "f' = differentiate f", and use it like you would normally in a mathematical context. These types of functions are called higher order functions. The title 'higher order' is the title given to functions which follow the lambda calculus representation of functions[**?**]. In lambda calculus allows functions to be partially applied below is an example of a function which squares the value passed as a paremeter in the purely functional language haskell.

Lazy evaluation is the result of not evaluating a functions result until its

result is needed. This means that arguments are not evaluated before being passed into a function, but only when their values are actually used, allowing functions to make use of infinite loops and ignore undefined values.

Hightlighting the differences between the Functional paradigm http://msdn.microsoft.com/en-us/library/bb669144.aspx

1.2.4 The Logical Paradigm

Logic programming is characterized by its use of relations and inference.[**?**] The logic programming paradigm influences have led to the 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 dfined in a "universal alphabet". Secondly, values are assigned to variables through atomatic substitutions called "most general unifiers", in some situations these assigned values may themselves be variables an are called logical variables [**?**]. Finally the control of a program in the logical paradigm comes from backtracking alone. Here lies 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 datatype [**?**].

1.3 Development Language

Based on the research the benifits of each programming paradigm it appears that the functional programming paradigm has more to offer a developer looking to produce a solution to a parallel problem.

Functional work differntly from imperrative Rather than performing actions in a sequence, they evaluate expressions

1.3.1 Erlang

Joe Armstrong[**?**] has emphasised, when talking about the language that he created, that robustness is a key aim of Erlang. The language was developed by the telecoms company Ericsson during a pursuit for a better way of approaching things. Safety and error management are very important environments such as telecoms as downtime is something to be avoided at all costs. Such is th focus on error handling in Erlang that there are three kinds of exceptions, errors, throws and exits; these all have different uses[**?**]. Proper use of Erlang's extensive error handling systems can be used to program robust applications. Another intersting 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 benficial for systems which ideally shouldn't stop such as, telephone exchanges or air traffic control systems.

Memory 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 becase 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 require little memory (lightweight), and creating and deleting processes and message passing require little computational effort [**?**]. These benifits 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 miliseconds.

With its strengths in reliability, error handlinhg and light weight concurrencty Erlang naturally excells at distributive programming. Erlang has no shared memory,

1.3.2 Haskell

Haskell is a purely functional programming language, this means a function will always return the same result if passed the same parameters; this is also known as 'determinism'????. Keeping code pure eradicates the possibility of many bugs which find their cause from a dependence on side effects the programmer assumed would be their but weren't. This kind of issue is increasingly likely to happen the larger an impure program grows, but Haskell is free of this problem.

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 sepcifically allocate and deallocate memory, which done wrong leads to several problems. But Haskell continues this abstraction to another fundemental 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 unecessary nondeterminism.[**?**]

The concept of a variable is quite different in Haskell to that of other languages. What might appear to be an assignment statement in other languages, in haskell translates to a name becoming bound to a value. This doesn't follow the usual abstracting concept of variables as a memory cell as in imperative languages. The way variables are treated in Haskell is similar to the way variables are treated in mathematics (once x is assigned a value during a calculation x is always that value). This form of assigning data is also called immutability and is also a feature of the Erlang language.

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 maybe combined with a function which conditionally increments a value, producing a function which updates a data datastructure in a concise yet flexible way. The concept of conciseness and elegance is often discussed within the Haskell community.

Non-strictness 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 might may have made the wrong decision concerning the type of a variable in memory. Haskells 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 doesn't 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 results in the majority of compiled programs always running as expected.

1.3.3 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 their program 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 func-

tionality 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 affects 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 that are not natively supported by the Java Virual Machine. For example the Java virtual machine lacks primitives in order to make stack frame re-use for tail calls efcient, therefore leading to Scala code sometimes compiling to a larger amount of byte and sometimes running slower than the equivalent Java[**?**].

1.3.4 Summary 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, and as such was learnt throughout the duration of the project because of the substantial amount of benifits the language provides in tackling the problem. It offers referential transparency through its purity allowing parallelism in alogrithms 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 seperate 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.

Non deterministic Parllelism

5 The speed of Haskell[**?**] Let me first state clearly that the following only applies to the general case in which speed isn't absolutely critical, where you can accept a few percent longer execution time for the benefit of reducing development time greatly. There are cases when speed is the primary concern, and then the following section will not apply to the same extent. Some of these features, such as closures, are likely to be supported soon, but many others never will. Therefore, a lean code at a high level of abstraction written in Scala can

be compiled to a large amount of bytecode, degrading runtime performance, as shown in this presentation.

To exploit a parallel processor, it must be possible to decompose a program into components that can be executed in parallel, assign these components to processors, coordinate their execution by communicating data as needed among the processors, and reassemble the results of the computation. Compared to traditional imperative programming languages, it is quite easy to execute components of a functional program in parallel [**?**]. Due to the property of referential transparencyand the lack of sequencing, there are no time dependencies in the evaluation of expressions; the nal value is the same regardless of which expression is evaluated rst. The nesting of expressions within other expressions denes the data communication that must occur during execution. [**?**]

Thus executing a functional program in parallel does not require the availability of a highly sophisticated compiler for the language. However, a more sophisticated compiler can take advantage of the algebraic laws of the language to transform a program to an equivalent program that can more efficiently be executed in parallel.[**?**]

In addition, frequently used operations in the functional programming library can be be optimized for highly efficient parallel execution. Of course, compilers can also be used to decompose traditional imperative languages for parallel execution. But it is not easy to nd all the potential parallelism. A smart compiler must be used to identify unnecessary sequencing and nd a safe way to remove it.[**?**]

1.4 Approaches to Parallelism

1.4.1 The Limits of Parralelization

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 speedup 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 benifiting 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 known formula below. The formula calculates the potential speedup 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 forumula 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 referrs 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). [**?**]

There are some models around which programmers may base a parallel algorithm around and similarly there are some data structures which .

1.4.2 Dynamic Multithreading

A Program with spawn sync and return statments A directed asyclic graph can be generated from the runtime of such a program where vertices are each [portion of execution up until a spawn/sync/return statment]

Two performance measures sufce to gauge the theoretical efciency of multithreaded algorithms. We dene the work of a multithreaded computation to be the total time to execute all the operations in the computation on one processor. We dene the critical-path length of a computation to be the longest time to execute the threads along any path of dependencies in the dag. Consider, for example, the computation in Figure 1. Suppose that every thread can be executed in unit time.

Then, the work of the computation is 17, and the critical-path length is 8. When a multithreaded computation is executed on a given number P of processors, its running time depends on how efciently the underlying scheduler can execute it. Denote by TP the running time of a given computation on P processors. Then, the work of the computation can be viewed as T1, and the critical-path length can be viewed as T. The work and critical-path length can be used to provide lower bounds on the running time on P processors. We have TP T1/P , (1) since in one step, a P-processor computer can do at most P work. We also have TP T , (2)

since a P-processor computer can do no more work in one step than an infinite-processor computer. The speedup of a computation on P processors is the ratio T1/TP , which indicates how many times faster the P-processor execution is than a one-processor execution. If T1/TP = (P), then we say that the P-processor execution exhibits linear speedup. The maximum possible speedup is T1/T, which is also called the parallelism of the computation, because it represents the average amount of work that can be done in parallel for each step along the critical path. We denote the parallelism of a computation by P.

Scheduling MIT http://www.catonmat.net/blog/mit-introduction-to-algorithms-part-thirteen/

threads introducing determinism[**?**].

1.4.3 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." [http://lampwww.epfl.ch/ odersky/papers/jmlc06.pdf] Message passing Actors can be creating new actors, decideing what action to take on a message or sending a message to another actor simulatneous to recieving a message. A common xxx in Erlang and Scala

1.4.4 Divide and Conquer

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. With a large MapReduce deployment (1000's of compute nodes) these steps to partition the work, compute something, and then finally collect all results is non-trivial. Things like load balancing, dead node detection, saving interim state (for long running problems), are hard problems by themselves.

1.4.5 Embarassingly Parallel

Embarassibly Parallel is the term given to problems where the
Other Approaches
Software Transactional Memory

1.5 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 this 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.

1.5.1 Networks simulations

http://www.cs.mcgill.ca/ carl/largescalenetsim.pdf

1.5.2 Molecular dynamics

Millions of atoms http://www.mcs.anl.gov/uploads/cels/papers/scidac11/final/jiang$_w ei.pdf$

1.5.3 Biological Simulations

Blue brain Fully implicit parallel simulation of single neuron experiencing linear speed up http://bluebrain.epfl.ch/files/content/sites/bluebrain/files/Scientific

1.5.4 Simulations for Games and Other Purposes

Colony Intel Employs SIMD to also insure instruction level parallelism http://software.intel.com/en-us/articles/vcsource-samples-colony/

parallel crowd simulation

1.6 Collision Detection

In a large scale simulation, it follows naturally that in denser areas of the simulation there will be large amounts of collision detection. In order to make an informed decision on how to tackle the collision detection problem research was carried out on various areas of collision detection and methods that may allow for parallelism.

Different methods

Why each method is effective and disadvantages.

Priori and Posteri / Discrete vs Continuous

Bounding Boxes various types [diagram] In addition to the bounding volumes covered here, many other types of volumes have been suggested as bounding volumes. These include cones [Held97], [Eberly02], cylinders [Held97], [Eberly00], [Schmer00], spherical shells [Krishnan98], ellipsoids [Rimon92], [Wang01], [Choi02], [Wang02], [Chien03], and zonotopes [Guibas03].

PartitioningBinary Space Partioning Bottom-up Construction Strategies When grouping two objects under a common node, the pair of objects resulting in the smallest bounding volume quite likely corresponds to the pair of objects nearest each other. As such, the merging criterion is often simplified to be the pairing of the query node with its nearest neighbor.

Locating the point (or object) out of a set of points closest to a given query point is known as the nearest neighbor problem. This problem has been well studied, and many different approaches have been suggested. For a small number of objects, a low-overhead brute-force solution is preferred. For larger numbers, solutions based on bucketing schemes or k-d trees are typically the most practical. A k-d tree solution is quite straightforward to implement (see Section 7.3.7 for details). The tree can be built top down in, on average, O(n log n) time, for example by splitting the current set of objects in half at the object median.

1.7 Visual Representation

OpenGL 2D

1.8 Definitions

1.9 Social, Ethical and Legal Issues

Before embarking on this project it was essential to look into any ethical issues that may arise throughout the course of the project or as a result of using the project's delieverable. So far none have no ethical issues have been encountered however ethics will be kept in mind throughout the course of the project and any issues will be documented in the project log during the project and the technical report at the end of the project.

(A critical appreciation of ethical issueds) A discussion of the licences that each library is provided under. Take note of: Intellectual property. Research ethics. Plagiarism.