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

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 one that inherently produces solutions capable of scaling well so that performance doesn't dip as 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."[?]

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

1.3.3 Scala

purely OO First, we wanted to be a pure object-oriented language, where every value is an object, every operation is a method call, and every variable is a member of some object. So we didn't want statics, but we needed something to replace them, so we created the construct of singleton objects. But even singleton objects are still global structures. So the challenge was to use them as little as possible, because when you have a global structure you can't change it anymore. You can't instantiate it. It's very hard to test. It's very hard to modify it in any way. [http://www.artima.com/scalazine/articles/goals$_o f_s cala.html$]

With the advent of multi-core processors concurrent programming is becoming indispensable. Scala's primary concurrency construct is actors. Actors are basically concurrent processes that communicate by exchanging messages. Actors can also be seen as a form of active objects where invoking a method corresponds to sending a message. The Scala Actors library provides both asynchronous and synchronous message sends (the latter are implemented by exchanging several asynchronous messages). Moreover, actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply. [http://www.scala-lang.org/node/242]

Of the challenges we were facing is we wanted to be both functional and object-oriented. We had very early on the notion that immutable objects would become very, very important. Nowadays everybody talks about immutable objects, because people think they are a key part of the solution to the concurrency problems caused by multi-core computers. Everybody says, no matter what you do, you need to try to have as much of your code using immutable objects as possible. In Scala, we did that very early on. Five or six years ago, we started to think very hard about immutable objects. It actually turns out that a lot of the object-oriented field up to then identified objects with mutability. For them, mutable state and objects were one and the same: mutable state was an essential ingredient of objects. We had to, in essence, ween objects off of that notion, and there were some things we had to do to make that happen.

interoperability with Java (can cause a lapse back to standard OOP programming)

Negatives Performance differences usually arise from the featf the challenges we were facing is we wanted to be both functional and object-oriented. We had very early on the notion that immutable objects would become very, very important. Nowadays everybody talks about immutable objects, because people think they are a key part of the solution to the concurrency problems caused by multi-core computers. Everybody says, no matter what you do, you need to try to have as much of your code using immutable objects as possible. In Scala, we did that very early on. Five or six years ago, we started to think very hard about immutable objects. It actually turns out that a lot of the object-oriented field up to then identified objects with mutability. For them, mutable state and objects were one and the same: mutable state was an essential ingredient of objects. We had to, in essence, ween objects off of that notion, and there were some things we had to do to make that happen.ures of Scala that are not natively supported by the JVM. 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.

1.3.4 Summary choice

This project will take 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 due to the benifits it offers in regard to the nature of the problem. Haskells type system is th richest and most expressive, it 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.

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 in nite-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/

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

an analysis.. What large scale systems are out there What do they do how do they do it effectively.

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

Molecular dynamics Millions of atoms http://www.mcs.anl.gov/uploads/cels/papers/scidac11/fin

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

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

1.6 Collision Detection

Large scale simulations

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].

Partitioning 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.

Binary Space Partioning

kDimentional Trees

1.7 Visual Representation

OpenGL 2D

1.8 Definitions

1.9 Planning

1.10 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 [**?**].

1.11 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 World
- Representing the Pheremone Levels
- Individual Ant Intelligence
- Collision Detection
- Parallelize Algorithms further
- Distributing Problem

1.12 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.

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.

1.13 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 git. 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.

1.14 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.