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 view will be limited to more realistic sensory field. With each ant making decisions based on 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 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 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 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 the research continues further, into methods mentioned in this discussion that appear most likely to provide a solution.

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 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 [**?**]. 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 pheremones can be used to communicate a range of messages but in general they are focused on marking paths to profitable food sources and signaling 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 and return with the food to it's colonies nest. An ant will also communicate other food locations to other ants in the colony through the use of pheremones, 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. While 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 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 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 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.

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 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.

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 discuss their strengthes and weakness, keeping the multicore 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 statments 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 interperated 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 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 also be tested seperately. 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 benifits 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 benifits 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 speedup obtained. But the encapsulation of data as an effect of an object-orientated language would favor 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 determinstic, 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 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 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 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 therfore aid in reasoning about parallelism and keeping data coherent as read-only datastructures can be passed to parallel processes and modifications that occur leave the old structure still intact.

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

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

Table 1: Table of Foos

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 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 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 datatype" [**?**].

Development Language Based on the research into the benifits 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 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 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 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 which 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,

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 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. Determinism isn't 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 affects, 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 hav-

ing 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 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 Virual 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 [**?**].

0.1.1 Summary and choice 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 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.

The Haskell compiler GHC supports multicore 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 multicore 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 chose 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 effected by the developers knowledge of bothe 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 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 used formula below.

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

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). [**?**]

Software Transactional Memory The approach taken in Software Transactional Memory (STM) can be seen as Task parallelism in this approach the tasks run on seperate threads which have to be explicitly envoked 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 multithreaded programming, by providing a software abstraction of the transactional memory technique used on multicore 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 with in 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 opperation simultaneously on a bulk of data. This allows for portentially 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 strurcture). Research and development is currently being undertaken for an extension on this approach in Haskell under the name Nested Data parallelsim [**?**]. 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 condititional 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 simulatneous to recieving 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 thread 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 complecity. 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 muliple processes are making an update to the buffer a thread safe implementation re-

quires 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 indpendent 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 thouasands 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 Embarassibly 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 embarassing parallisation.

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 queing 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 doesn't 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 speedup 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 partioning 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 [**?**].

0.1.2 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 distribute. Employs SIMD to also insure instruction level parallelism [**?**]

Dependencies of a taskset are previously created tasksets that must complete before the new taskset can begin execution. The simulation is partitioned such that each taskset does not require data calculated in other tasks within that operation, and that the range of data being operated on is easily split up. Our update function performs the following parallel steps, operating on arrays where the start and end indices demarcate a group of tasks executed in parallel across the cores of the CPU.

1. Assign units to bins 2. Determine new directions for binned units (depends on number 1) 3. Update units along their new direction and perform unit logic (depends on number 2)

A parallel crowd simulation produced by the research team at Sony [**?**],

The work being done on the cell processor - "buckets" lattices and nearest n.

Collision Detection In a large scale simulation, 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 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.

nav mesh

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.

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 would which is easier to comprehend. OpenGL 2D

0.2 Definitions

0.3 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.