0.1 Implementation

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

[small diagram]

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

[small diagram]

The former solution was pushed back due to a lack of understanding Monads, the Random Monad would have been required to develop this solution making use of the Prelude module, System.Random (The Prelude is the name given to Haskell's standard library). So things pressed on with the second approach to the problem. While writing this decision making function to split the list of Ants into a list of lists it became clear that this was not the best approach to take as parallelism would be limited and especially calculating the concentration of a pheromone at a specific location would require a lot of processing which would introduce a lot of dependence. Use of a non-grid based system was the main issue when looking at the complexity of calculating pheromone concentration at a specific location. This stems from the fact that an instance of pheromone may have a location which does not necessarily correspond exactly to an ant's location but is close enough to affect it. An algorithm to calculate the exact concentration of pheromone at a specific location would need to check every pheromone instance in the pheromone list and to see if the location of the ant falls within the radius of the pheromones effective area. This would involve excessive and unnecessary checks which would fail as every Ant would have to be checked against every instance of element in the list of pheromones.

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

sqrt(sqr(m) + sqr(n)) (1)Where m is the difference between the two x locations and n the difference between the two y locations. This would be then be compared to the sum of the two radii, be it ant ant or ant phereomone, if the sum of the radii is less than the result of the forumla above then the move cannot take place.

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

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

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

The problem of limiting all these pair wise checks between every ant pheremone instance and food location was not the focus of this project, however would make a great consideration for future work as this would lead to a more liberated and naturalistic simulation. To resolve this problem the decision was taken to limit movement of ants and locations of pheromone instances to a grid based structure, this provided a way for checks to be performed on the pheromone list in order to determine whether or not a certain amount of pheromone was at specific location. In order to reduce the amount of checks multiple lists were employed to represent each row within a grid for both ants and pheromones. With every row containing the number of elements as columns within the world there was a requirement to store the absence or presence of data at a specific location within the grid. This was done through the use of Haskell's parametric data type Maybe which can either hold a Just value or Nothing. This enables functions to be written which would accept the absence of an ant as a valid type as well as valid ant information so long as they were wrapped in the algebraic type Maybe. Thus, abilities such as mapping a function over the data representation of the simulation world could be preserved.

Until this point in the project Ants had been stored as a parametric data type which accepted (The parameters for an ant and their types) in order to retrieve values from them, functions needed to be programmed to retrieve specific values from data of an ant type. To retrieve values from a parametric data type

Haskell's process of pattern matching is employed by using it to bind the values of the parameterised type to variables and then by providing the variable as the result of the function it is possible to retrieve the desired value from the data type. This is also known as deconstruction [?].

```
data Ant = Ant Location Direction
         deriving(Show)

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

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

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

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

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

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

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

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

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

```
map' :: (a -> b) -> [a] -> [b]
map' _ []                = []
map' function (w:[])          = [function w]
map' function (w:x:[])        = fx `par` [function w,fx]
     where fx = function x
map' function (w:x:y:[])      = fx `par` fy `par` [function w,fx,fy]
     where fx = function x
           fy = function x
map' function (w:x:y:z:zs) = fx `par` fy `par` fz `par`
                               (function w:fx:fy:fz : (map' function zs))
     where fx = function x
           fy = function y
           fz = function z
```

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

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

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

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

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

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

This is modelled implemented in Haskell code with the function below.

```
antScenarios qs pos currAnt adjAnt
       | (isNothing currAnt) && (isNothing adjAnt) = procEdgeAntAtNode (1+pos) qs
       | (isJust currAnt) && (isNothing adjAnt)    = loneEdgeAnt qs True pos
       | (isNothing currAnt) && (isJust adjAnt)    = loneEdgeAnt qs False pos
       | (isJust currAnt) && (isJust adjAnt)       = doubleEdgeAnt qs False pos
```

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

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

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

In the process of tackling this problem it was discovered that debugging Haskell code is also very different from solving bug in imperative code. A runtime debugger that tracks a variables value is almost redunant as variables can hold 'thunks', unevaluated values, due to Haskell's property of non-strictness. It was discovered that a simple way to evade this problem is to break large functions down into smaller ones and monitor the return values of each of these functions closely using ghci, the interactive Haskell console. However, in this instance this approach only allowed me to narrow down the problem of the infinite loop to the function procEdgeAntNode. Further research into the issue introduced a program for code coverage called HPC (Haskell Program coverage tool). The program highlights the functions called during the running of a function and aided in confirming that the situation was truely an infinte loop. It was not until the decision was made to enable all warnings on compilation with the command line option -Wall being passed to the compiler that the vast amount of potential program problems were brought to light.

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

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

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

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

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

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

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

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

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

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

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

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

0.1.1 Testing

As mentioned during the research section, there is a haskell library called quickCheck which provides the ability to run algebraic tests on functions produced in Haskell. The quickcheck properties were generated for the GraphOps.hs module, initially there were issues with the test running but never reaching completion. It was soon realized that the values quickCheck generates for it's tests needed to be restricted in order to supply the functions with realistic valid data. For instance, checking a generated list's length was equal to the given parameter squared, with larger intergers ends up with this the test calculating the length of extremely lengthy strings causing the test to hang. Also, the a function might only work on positive integers, quickChecks automatic generators which provide random data given the type do not offer this flexibility. Therefore, custom generators had to be programmed.

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

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

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

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