Acknowledgements Background and Research and Context

- Simulations

A simulation is a genre of game, a life simulation game could revolve around a particular character and its relationships with other things within the simulation, or it could be a simulation of an ecosystem (Spore). Biological simulations may allow the player to experiment with genetics, survival or ecosystems, this can often be for educational purposes. Unlike other genres of games, simulation games do not usually have a set goals that allow a player to win the game. Rather they focus on the experience of control, wheter it be the lives of people, when micromanaging a family (The most notable example of this is Will Wright's The Sims.) to overseeing the rise of a civilization or success of a business.

Outside of games, biological simulations can be used for more than educational purposes, and are utilized by researchers, to test theories, which in practise would be impossible to carry out because of technological or financial constraints, and show their findings visually. These are often large scale simulations which demand large anounts of processing power due to the nature of the problems. for example the Blue Brain Project http://www.newscientist.com/article/dn7470–
mission-to-build-a-simulated-brain-begins.html and weather prediction http://en.wikipedia.org/wiki/Climate$_{m}od$

LINKING PARAGRAPH!!!!!!!!!!

- Multicore Programming

Processor clockspeeds aren't increasing at the rate they used to, with the [GPGPU cite!]. However Moore's law is still in tact as the processors are consisting of more cores. In order to harness the full potential of these multicore processors it is necessary to run computations on all the processors available cores. While the splitting up of work between a processor's cores might be a trivial process in theory. In practice it opens up several new issues which a programmer must address; into how many pieces should the computation be split, will the smaller computations need to communicate their results to other computations, is it necessary for some computations to finish executing before others begin? These are all questions which face a programmer when designing an application to make use of a multicore processor.

There are two words which occur often in discussions about multicore programming and these are Parallelism and Concurrency. It is important to clarify these terms:

Parallelism: Its a property of the machine where software is to be run. The application runs on multiple processors/cores, with the hope for faster speed than on a single processor machine.

Concurrency: Software is composed of, possibly unrelated, computations and multiple threads of control making effects to the world, via arbitrary interleaving. The expected results are therefore, non-deterministic because the total effect of the program may depend on the particular order of interleaving at runtime. Computation is based on this nondeterminism. [Nino and RWH]

Where concurrency is utilized the programmer has the responsibility of controlling the non-determinism using synchronisation, to make sure that the program runs as it is supposed to regardless of the order of execution. This is not

an easy task because theres no reasonable way to test that you have covered all the cases. ¡testing threaded programs cite!¿

In order to make programs run faster on parallel hardware, it is not neccessary to have concurrency. Its important that this issue is well understood if were to find a way to enable everyday programmers to use multicore CPUs.

-Functional programming as opposed to other paradigms

A functional language is language in which computation is carried out entirely through the evaluation of expressions. [Evolution of Functional hudak] Burge back in 1975 suggested the approach of evaluating function arguments in parallel, with the possibility of functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation [W. H. Burge. Recursive Programming Techniques. Addison-Wesley, 1975.]. ¡??Berkling also considered the application of functional languages to parallel processing??¿ [K. J. Berkling. Reduction Languages for Reduction Machines. In 2nd. Annual ACM Symp. on Comp. Arch., pages 133-140. ACM/IEEE 75CH0916-7C, 1975.].

¡!Definition of purity!¿ To be functionally pure, a method must satisfy two critical properties First, it must have no side effects. For a computational method to be free of side effects, its execution must not have any visible effect other than to generate a result. A method that modies its arguments or global variables, or that causes an external effect like writing to disk or printing to the console, is not side-effect free.

The second property is functional determinism: the methods behavior must depend only on the arguments provided to the method. The method must return the same answer every time it is invoked on equivalent arguments, even across different executions of the program [Verifiable functional purity in java Matthew Finifter Adrian Mettler Naveen Sastry David Wagner 2008]

Due to the absence of side-effects in a purely functional program, it is relatively easy to partition programs into sub-programs which can be executed in parallel: any computation which is needed to produce the result of the program may be run as a separate task. There may, however, be implicit control- and data- dependencies between parallel tasks, which will limit parallelism to a greater or lesser extent. [Parallel Functional programming an introduction Kevin Hammond] http://www-fp.dcs.st-and.ac.uk/ kh/papers/pasco94/pasco94.html

¡!!needed? Higher-order functions (functions which act on functions) can also introduce program-specific control structures, which may be exploited by suitable parallel implementations, such as those for algorithmic skeletons (Section 3.2).!!¿

Here is a classic divide-and-conquer program, a variant on the naive Fibonacci program. Since the two recursive calls to nfib are independent, they can each be executed in parallel. If this is done naively, then the number of tasks created is the same as the result of the program. This is an exponentially large number ().

There are many ways to exploit the possible parallelism present in a functional program. ¡¡¡Most systems have selected a set of these, and it is therefore difficult to isolate the effect of a single technique on overall performance, even when concrete performance results are available.¿¿¿

-Haskell's Benifits -The problem

¡¡¡ In Chapter 2 important concepts in parallelism in the last decade and relevant publications are reviewed. Studying them provided insights about the inherent problems in the development of large scale simulations and their possible solutions.

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

Chapter 4 explains and justifies the architectural and simulation design choices that were made.

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

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

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

Requirements

The requirements therfore on this project are to produce an ant colony simulation which exploits the parallel benifits of the purely functional programming language Haskell. The behaviour of the project's resulting artifact aims to portray an functioning ant colony. Ants should seek out food using exploratory strategies, communicate locations of food to other ants in the colony using pheremones and return to the colony's nest. The ant colonies increasing size should be reflected in the size of the nest, and the nests size in turn will dictate the amount of new ants generated by the colony. The nest's physical size will grow relative to the amount of food successfully brought back to the colony by the ants in the simulation.

Exploratory Strategies Ants will explore the simulation world by moving randomly away from the nest, each ant will have a direction property which when set allows them to track their location relative to where they have already been. On example of the use of storing directions within the ant is to detect Moving away from other ants with no food.

Communicating locations of food Leaving a trail of pheremone. Other ants can then pick up on this trail to more efficiently find food.

Returning to the colony This requirement could also be achieved through the use of pheremones, or by allowing ants to retrace their steps.

Nest size The nest size will increase relative to how much food is brought back to the nest. The ants task of bringing food back to the nest enables the colony to fulfill its task of increasing in size.

Above all the aim of the project it to exploit the parallelism exposed by pure function in Haskell. The project could t

Test driven development

Algebraic testing

Design

Desgin techniques

Unlike imperative languages such as Java and C++ Haskell does not use classes to create objects and then send messages between objects. Rather it uses Declarative and Data-driven methods to process data. Haskell doesn't strictly speaking do any of these; because it is the programmer that does things using Haskell. This makes it more natural for imperative or Object Oriented programmers to take a different approach when programming in Haskell. It is critically important to use the right technique or tool for the job at hand.

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

The classes used by Haskell are similar to those used in other object-oriented languages such as C++ and Java. However, there are some significant differences:

Haskell separates the definition of a type from the definition of the methods associated with that type. A class in C++ or Java usually defines both a data structure (the member variables) and the functions associated with the structure (the methods). In Haskell, these definitions are separated. The class methods defined by a Haskell class correspond to virtual functions in a C++ class. Each instance of a class provides its own definition for each method; class defaults correspond to default definitions for a virtual function in the base class. Haskell classes are roughly similar to a Java interface. Like an interface declaration, a Haskell class declaration defines a protocol for using an object rather than defining an object itself. Haskell does not support the C++ overloading style in which functions with different types share a common name. The type of a Haskell object cannot be implicitly coerced; there is no universal base class such as Object which values can be projected into or out of. C++ and Java attach identifying information (such as a VTable) to the runtime representation of an object. In Haskell, such information is attached logically instead of physically to values, through the type system. There is no access control (such as public or private class constituents) built into the Haskell class system. Instead, the module system must be used to hide or reveal components of a class. (Gentle introduction to Haskell Version 98)

Data types in Haskell and then quick checks [Method::Type-¿ Type -¿ Type - Method x y = undefined - quickCheck] – reference

QuickCheck is a combinator library written in Haskell, designed to assist

in software testing by generating test cases for test suites. It is compatible with the GHC compiler and the Hugs interpreter. In QuickCheck the programmer writes assertions about logical properties that a function should fulfill; these tests are specifically generated to test and attempt to falsify these assertions. The project was started in 2000. Besides being used to test regular programs, QuickCheck is also useful for building up a functional specification, for documenting what functions should be doing, and for testing compiler implementations.[1] Re-implementations of QuickCheck exist for C,[2] C++,[3] Chicken Scheme,[4] Clojure,[5] Common Lisp,[6] D,[7] Erlang,[8] F,[9] Factor,[10] Io,[11] Java[12][13] JavaScript,[14] Node.js,[15] ObjC,[16] OCaml,[17] Perl,[18] Python,[19] Ruby,[20] Scala,[21] Scheme,[22] Smalltalk[23] and Standard ML.[24]

It is generally agreed that formulating careful specifications of software is a good idea, whether before or after the software is written. Specifications improve software design, by revealing unclear behaviour or complicated cases. Specifications guide programming, by offering an objective criterion for whether a program is right or wrong. Specifications assist reuse, by documenting exactly what software is supposed to do. However, specifications are used much less in practice than one might expect. Our goal in developing QuickCheck is to add value to specifications, by offering a short-term payoff for their use.

QuickCheck encourages you to formulate precise, formal specifications, in Haskell, a language you already know. Like other formal specifications, QuickCheck properties have an unambiguous and clear meaning. QuickCheck checks your program against the specification, by testing. Although this cannot guarantee that the program and specification are consistent, it greatly reduces the risk that they are not. It is easy to recheck consistency after every change to a module. QuickCheck specifications document how you tested your program; other programmers using your code can see which properties have been tested, and which have not. QuickCheck reduces the time spent on testing, by generating many test cases automatically. This saving can be set against the time spent on formulating the specification in the first place.

UML is not limited to OOP. Even though it's founded in OOA/D it can be adapted a long way.Your data structures are typically determined by the operations you want to do on them, so it depends on the kind of UML model do you wish to make

Parametric polymorphism's also remarkably effective at helping strip out the irrelevant

Predicted code flow of the application On the function/method level. More detailed than that.

High level parallel design (pmap) discussion. 'par 'seq' annotations and other forms of denoting parallelism.

Data flow

Representing Source code

(Talk about the non parallel Then the parallel version

Split into Epochs of design

Mind map)

Representation of the Simulation world - Split into Epochs of design

The design of the simulation went through many iterations.

The list based Worlds design free movement

List based Worlds location tags

Graph based World Hold all

Graph based Worlds non parallel

Graph based Worlds parallel

The initial concept for the representation of the simulation world was list based. Several lists would be produced for everything in the world, for an example a list of Ants currently in the simulation, a list of Food currently in the simulation and a list of the surfaces in the simulation. Each element in the lists would hold a location value and

Types The types that will be involved in modeling the simulation of an Ant colony are as follows.

Ant :: Ant Pheremone :: Double Food :: Double Nest :: Bool AntQuadrant :: Graph -¿ Holds- Possible Ant PherQuadrant :: Graph -¿ Holds- Pheremone Level FoodQuadrant :: Graph -¿ Holds- Possible Food NestQuadrant :: Graph -¿ Holds- Possible NestArea AntWorld :: Graph -¿ Holds- Many Ant Quadrants PherWorld :: Graph -¿ Holds- Many PherQuadrants FoodWorld :: Graph -¿ Holds- Many FoodQuadrants NestWorld :: Graph -¿ Holds- Many NestQuadrants

Key Functions The functions responsible for the flow of the program are

stitchUpAntQuads - Sequentially processes pairs of edge-nodes on a Graph (to possibly move Ants between AntQuadrants). stitchUpPherQuads - Sequentially processes pairs of edge-nodes on a Graph (to manage the spread of Pheromone between PherQuadrants).

processAntQuadrant - Moves each Ant within a quadrant. processPherQuadrant - Evaporates the Pheremone

Main Given Starting parameters create a World. -Parameters Size of Quadrant :: Int Size of World :: Int Amount of Ants :: Int Amount of Food :: Int

-Initialize random nest area proportional to the amount of Ants. -Randomly position Ants around the Nest Area -Randomly position Food (not In Food Area) -Initialize all PherQuadrants as 0 except for Food Quadrants which maintain 10.0

Loop

In parallel -stitchAntQuads -stitchPherQuads

In parallel - processAntQuads - processPherQuads

Display

ReQuading Graph Density calculating densest points NP-hard?

When graphs are split at densest points More serial calculations.

Implementation

This chapter will discuss what has been done so far in terms of implementation and the problems that appeared in the process.

Started off by implementing a Graph in Haskell. Difficulties with the Graph api. 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. Discussed further in the conclusion, (Looking back I 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.)

The project then progressed by way of finding ways to modify the graph data structure, because data in Haskell is persistent this means creating new data of the graph type as there is no assignment. The first function to be developed was swapping nodes on a graph.

The progression of Feature sets

Representing the Ant (From a Parametric type -¿ To a Record)

Wrapping the Graph class (GraphsOps) Ant storing Ants in Graph Quadrants World -¿ Storing Quadrants Stitching Up Graphs Printing Ant Quadrants for testing -$^{S}erialModeProcessingstepsandparallelismThreadscopedebuggingPheremoneQuadrantPrintingan$
$Getsstuckincorners)IntroductionofFoodQuadrantsfortestingExpandingAntBehaviour(Exploration(Random$
$arallelmode$

Implementing Client Server (Remote. Cloud Haskell) Haddock GPU acceleration Quadrant splitting - Naievely Markov clusters - Graph averaging. -Distrubuted mode

GUI - wxcore fail Run simulation based on Settings

Developed to move nodes based on other values.

Stitching together the Graphs.

Developing the process loop. Parallel Strategies.

Positioning new Ants in the world.

Positioning food locations.

Debugging http://www.haskell.org/haskellwiki/Haskell$_{p}rogram_{c}overagehttp$:
$//www.haskell.org/haskellwiki/DebuggingInfinite_{l}oopshttp : //stackoverflow.com/questions/2861988/hask$
$defaulting - constraints - to - type$

Conclusion and Future Work

There have been many lessons learned over the course of this dissertation project.