Parallel Design Patterns II

B138813

February 2019

1 Introduction

We present our report for the second submission of the Parallel Design Patterns Coursework. We first discuss our implementation in Section 2, and demonstrate its correctness in Section 3. Although our implementation has some features similar to a framework, we were not able to succesfully seperate policy and mechanism to an adaquete extend. We will discuss this issue further in Section 4.

2 Implementation

We implemented our squirrel simulation in C++. We present a UML diagram of our implementation in Figure 1. To maximise code re-use, we abstracted as much common functionality into the parent actor class as possible. The actor class holds methods such as `send_msg()` and `msg_recv()` which act as wrappers to MPI functions, whilst also providing extra functionality. For example, `msg_recv()` returns a three item tuple, which contains:

- A boolean, which indicates if the message was succesfully recieved.
- An integer, which indicates where the message was recived from.
- An integer, which is the message itself.

The semantics of the message integer are encoded into the `MSG` enum, which is shared by all actors. This standardisation makes it easy for the programmer to reason about messages, allowing them to dictate how a message if handled in each child class's main event loop. The function `send_msg()` is simple convenience wrapper around `MPI_Bsend()`. Each class inherits directly from the actor class, except for the controller, which is a child of the master class. We made this decision because the master and controller share most of their functionality and variables. The master class starts the simulation, and then provides process pool functionality. The controller class keeps track of the number of live and infected squirrels.

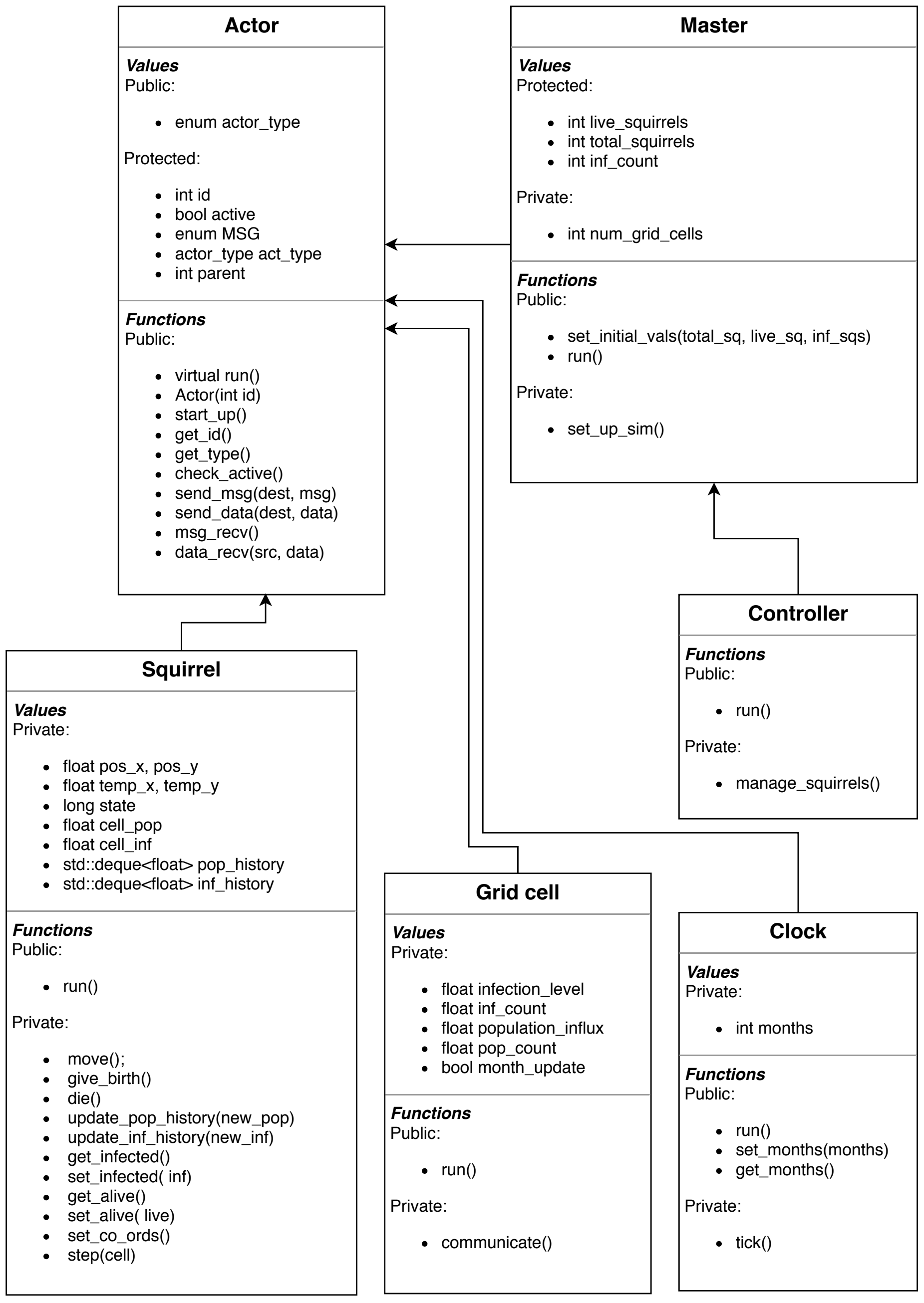

Figure 1: UML Diagram of the Program

Both classes need to know the simulation's initial squirrel numbers, and share functions and values to implement this. If C++'s inheritance model did not allow both child and grandchild classes to implement their own versions their parent's virtual functions, this pattern would not be possible. The squirrel class has the most functions of any class, partly because of how active the biologists model demands the squirrel to be, and partly because of how we have decided to model our actors.

Each actor has an actor type, as designated by the enumerate type `actor_type`. There three different actor type for squirrels; infected squirrels, dead squirrels and normal, healthy squirrels. To safely change a squirrel from one of these types to another, we thought it best to encapsulate the code within a function. In doing so, we keep our code compliant with object oriented design philosophies, but diverge from high performance programming ones. However, we do not think the addition of these functions degrades the performance of our work too much, and improves the readablity of our code.

There are four key communications between our actors. We have endevored to abstract them and encapulate them into our actor class. We will now describe our implementations of those actors.

2.1 Actor Start

2.2 Clock tick

We have made use of a clock actor to send messages to every grid cell every month, modelled as a period of 0.2 milliseconds. We refer to this time period message as a tick. In the grid cell's `run()` function, while it is active we call the communication function. This

2.3 Squirrel Step

2.4 Squirrel Birth

3 Correctness

If no squirrels are infected, the population will continually grow.

4 Further Work

Further generalise data recv or make a blocking data recv for location vector.