Parallel Design Patterns Coursework Part 2

Exam Number: B136013

March 26, 2019

Contents

1 Project Description

This document contains the coursework report for parallel design patterns course. Generally, it describes the design and implementation overview of the squirrel parapoxvirus disease simulation. Firstly, there is a brief introduction to the technology and tools used, then the developed applications are discussed. Both the framework and simulation are analysed concerning the architecture design, components functionality, scalability, performance and usage. The next section contains the testing approach based on the non-deterministic results, and finally, there is a discussion about simulationâĂŹs output.

2 Technology and Tools

Technologies For the development of this project, the programming language is C++. C++ is a low level, generic-purpose language, suitable for simulations that need to extract high performance out the hardware. The main reason why C++ has been chosen over C is because of the Object Oriented Programming (OOP) functionality. Unlike C, C++ incorporates the concepts of inheritance, encapsulation and polymorphism that are used in the application. In terms of the communication layer, OpenMPI (MPI) is selected. In this way, passing messages across different processes is a customised procedure that fits the needs of the application.

Building Tools When it comes to building tools, the choice of the appropriate compiler has to be based on the characteristics of the application. Knowing that this a C++ with MPI program **mpicxx** is a viable option, keeping in mind that the underlining platforms provide it. Accompanying the compiler flags are applied such as **-g**, **-std=c++11**, **-Wall**, and **-O2**. Finally, to coordinate the compiling and linking procedures, GNU Makefile is used for the solutions.

Platform The target hardware to execute the simulation is Cirrus. The application contains the appropriate **PBS** file to submit the job to the backend and get the results of the simulation.

3 Actor Framework

The application is separated into 2 different solutions. One is the actor framework that is analysed in this section, and the other is the specific simulation. The policy that separates the two programs is that the actor framework has not any knowledge about specific simulation mechanisms like squirrels or cell and all the actions that are related to them.

It just provides the underlying structure to create, kill actors and communicate with them. On the contrary, simulation is specific to the problem restricted to use the offered functionality of the framework in a specific and well-defined way. Generally, the actor framework is a modular system that can be applied to different applications without any restrictions as long as they implement actor parallel design pattern's concepts.

3.1 Architecture

3.1.1 Algorithm Strategy

The selected approach of implementing the actor framework is to assign more than one actors to the available units of execution (UE). In this way, the framework achieves a logical separation between the number of actors and the available number of processes. There is no doubt that this choice of strategy is more difficult than the 1-1 match of actors and processes. Despite that fact, this choice has been made because many benefits can be gained in terms of scalability and performance.

Scalability The selected approach is designed to provide high scalability. Based on the fact that the number of actors is independent of the number of processes, the application can expand, containing many actors without the need for many processes to run as well. This is the main reason that the specific algorithm strategy is selected.

Performance In terms of performance, there are advantages but also drawbacks. On the one hand, running experiments can help the developer decide how many processes should be used according to the specific simulation to extract the best available performance. Moreover, having in mind that the actor framework is a dynamic system which creates and kills actors at will, this approach increases the performance. This happens because the overhead to manage actors is minimised to allocate and free memory not to start and stop processes. On the other hand, as mentioned above the complex implementation reduces the performance slightly. The additional layers that affect that are a mapping between actors and processes and a messaging mechanism to redirect messages to the appropriate actors in UEs.

3.1.2 Supporting Structure

Under the above mechanism, a master-worker architecture is established. The specific architecture choice is made because of the need for a coordinator which is explained in the following paragraphs.

Master Master is a unique process that collects vital data for the simulation. Briefly, the master process is responsible for a variety of tasks. These are:

- Finding the next worker for new actors
- Notifying the workers when to start or end the simulation
- Spawn new actors to the appropriate workers
- Keep booking about the number of spawned and dead actors

Worker Workers are the processes that host actors. Their obligations are:

- Manage the data structure with the actors (add, remove, find)
- Execute the appropriate function depending on their state
- Transfer messages to the appropriate actor

3.2 Software Components

Actor Actor is an abstract class that encapsulates the necessary functionality for the possible derived objects. Moreover, it is designed to execute a user defined finite state machine (FSM), which is different for each actor type. In general, it provides the options to:

- Register compute functions (based on state)
- Register parse message functions (based on state)
- Create new actors
- Kill itself
- Send messages to other actors

Message Message is the core structural unit of communication between actors. It is designed as an array of float numbers. The fits array 17 elements. The size can be easily changed through the actor framework. The framework has allocated the first 8 elements for its communication. Beyond that, the user can define the data that wants to pass through the actors. 16 out of 17 elements are meant to be data. The first one specifies the command of the message. Again number from 0 to 4 are allocated by the framework. After that user can define simulation specific commands.

Messenger Messenger is the only mechanism responsible for sending and receiving messages that defined above. Both actors and processes communicate through this entity. It provides a set of functions that use MPI commands. The MPI functions are **MPI_Bsend**, **MPI_Send** and **MPI_Recv**. Due to the dynamic nature of the actor framework data regularly change. This is the reason why **MPI_Bsend** is used mainly by the framework.

3.3 Decoupling

Originally, framework and simulation developer together. Gradually, the functionality became to separate between these two entities. Performing regular code refactoring the split was achieved. One of the most challenging parts was that the user had to define functions which for example spawn new actors. Injecting callback functions carefully helped to achieve the goal of a modular design.

3.4 Usage

The final product of the actor framework is a dynamically linked shared object library; in other words, a .so file. Besides, a C++ header file has been generated that exposes an interface for users that want to develop new simulations. Finally, linking the user's simulation with this file can create the final executable that applies the actor pattern.

4 Simulation

4.1 Architecture

Actor design pattern is an asynchronous concept but for actors to interact with each other restrictions in their behaviour have to be added. For this reason, an FSM mechanism is the core implementation of how an actor behaves. Examples of how FSM function will be explained in detail during the presentation of simulation's actors.

4.2 Simulation Responsibility

The simulation developer has to use the framework in a pre-determined way to work as expected. There are some system requirements that the user has to meet. These are:

- Define actor types
- Define simulation commands
- Define simulation message data positions
- Feed the pattern with the necessary callbacks for initialising and spawning actors
- Use Messenger and Message objects for communication

Any misuse or omission of the frameworks exposed functions leads to the failure of the simulation.

4.3 Actors

Simulation actors are the classes that inherit the attributes of the previously mentioned actor class in the framework application. Actors are flexible to store any data they find useful or add functionality without restrictions. Nevertheless, there are some contracts that they need to serve. First, the constructor of the Actor has to be called in the indicated way containing data such as the new actor's id. Moreover, actors have to define and register their states and the function to be called on computing or receive message time.

4.3.1 Clock

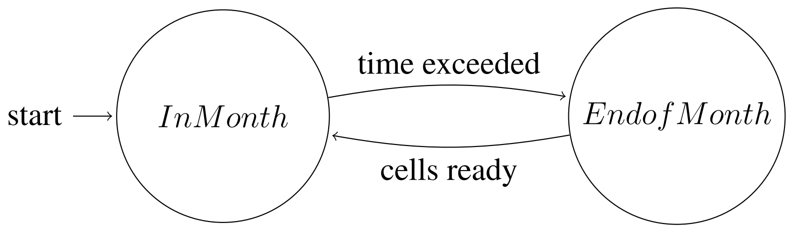


Figure 1: Clock FSM

Clock is the coordinator actor of the simulation. Its primary role is to provide awareness of time to the cells and the user. According to Figure 1 clock starts in state **In Month**. It uses the system's clock and every 1000 milliseconds (user-defined time) it changes the month moving to state **End of Month**. At this point waits for all of the cells to respond with their population influx and infection level. Then once again changes to state **In Month** and starts the next month. Clock receives messages from squirrels as well. Each time a squirrel is born or gets infected sends an appropriate message to the clock. In this way, the clock keeps statistics for the squirrels to provide to the user. These actions do not change the state of clock. The simulation can stop only at the end of a month. The conditions are:

- The current month is the last one (the user defines max month)
- More than 200 squirrels are alive

At the end of the simulation, clock writes the saved data to output files. These data are:

- Population influx
- Infection level
- Alive squirrels
- Infected squirrels

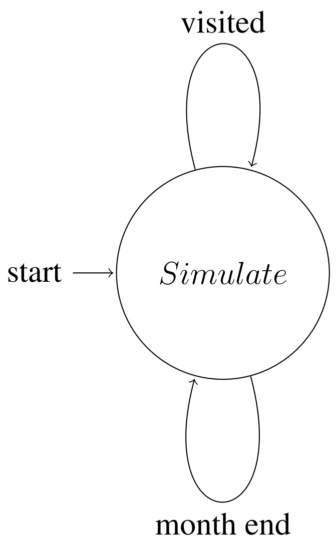


Figure 2: Cell FSM

4.3.2 Cell

Cell is an actor type that represents a region of land in the landscape. The current simulation supports only a 4x4 landscape, but it can comfortably adjust to more complicated shapes with appropriate modification to the code. As shown in Figure 2 cell is constantly in **Simulate** state. As mentioned above, it receives messages from the clock when the current month has finished. Then it sends to clock the population influx and infection level of the cell. Squirrels can also interact with the cell. When a squirrel visits a cell, it receives squirrel's health status to update population influx and infection level accordingly.

4.3.3 Squirrel

Squirrel is the actor class that this simulation studies. Squirrels start in **Live** state according to figure 3. Every 50 milliseconds a squirrel decides to act with a probability of 33% independently of the clock actor's state. This means that it takes approximately (Month duration/Duration to decide)*(possibility to act) = (1000/50)*33% = 7 moves per month. When a squirrel acts it does three things:

- Decide if it gives a birth
- Decide if it dies
- It takes a move

When a squirrel gives birth, it informs the master process indirectly and the clock actor directly to take care of the procedure. When it dies, based on the given algorithm. It informs once again the clock and the worker that it belongs to remove it. Then it changes its state to **Died**. Squirrels take moves. Each time a squirrel moves, first finds the target cell, then it sends its health status and goes to **Interact** state. At this point, it waits for the response of the cell about population influx and infection level. The time it gets the response it changes its state back to **Live**.

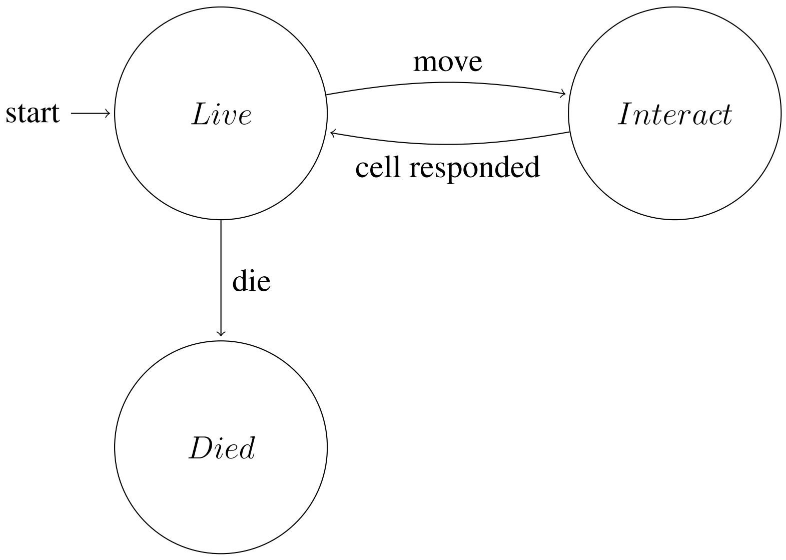


Figure 3: Squirrel FSM

4.4 Parameters

Simulation parameters are passed as arguments in the executable. In this way, the application is compiled once and can run with different inputs described in a bash script. The input parameters are:

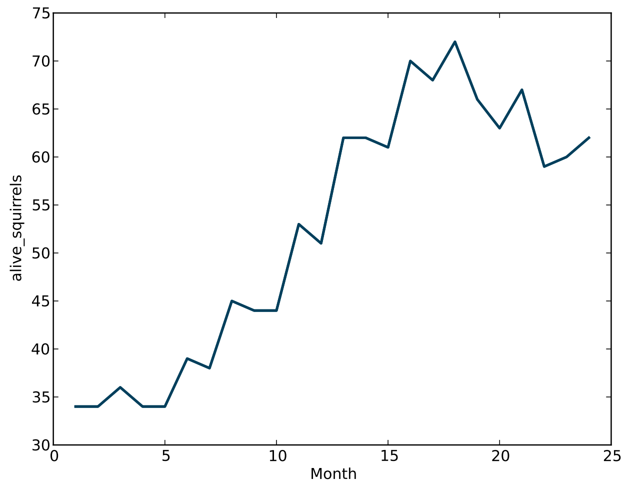
- **Squirrels** number of initial squirrels
- **Infection level** number of infected squirrels
- **Max months** number of simulated months
- Max squirrels number

Despite these arguments, there are some parameter variables defined in the **simulation.h** file. These values are supposed to be immutable. The reason is that the implementation of the simulation is bind to these values and there is the need for code modifications in order to make the input arguments. These variables are:

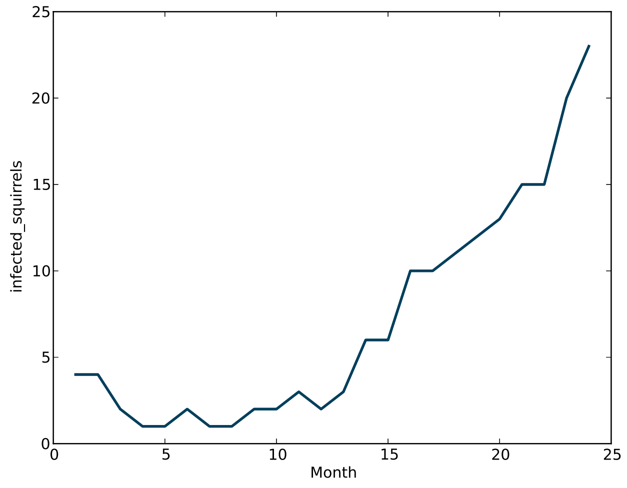
- Cells number
- Clocks number

5 Results

There are scripts designed to run the simulation and plot the results. Their usage is analysed in the **README.md** file. In this section, the results of a representative run are shown in Figures 4 (a), (b) and 5 (a), (b). The input parameters were 34 squirrels, 4 of them were infected, the maximum number of supported squirrels is 200, and the simulation lasts 24 months.

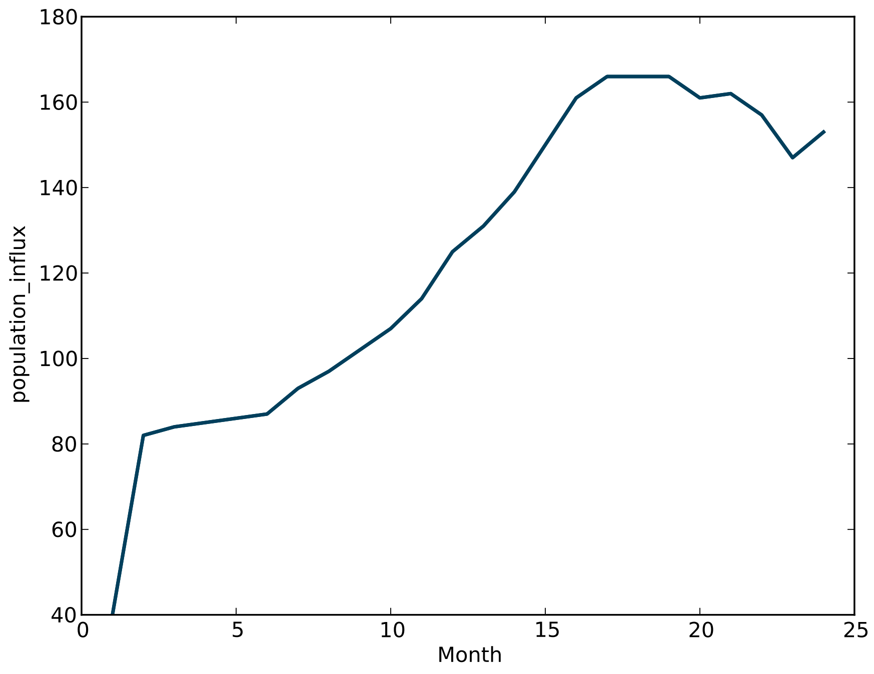


(a) Number of alive squirrels in the landscape for each month

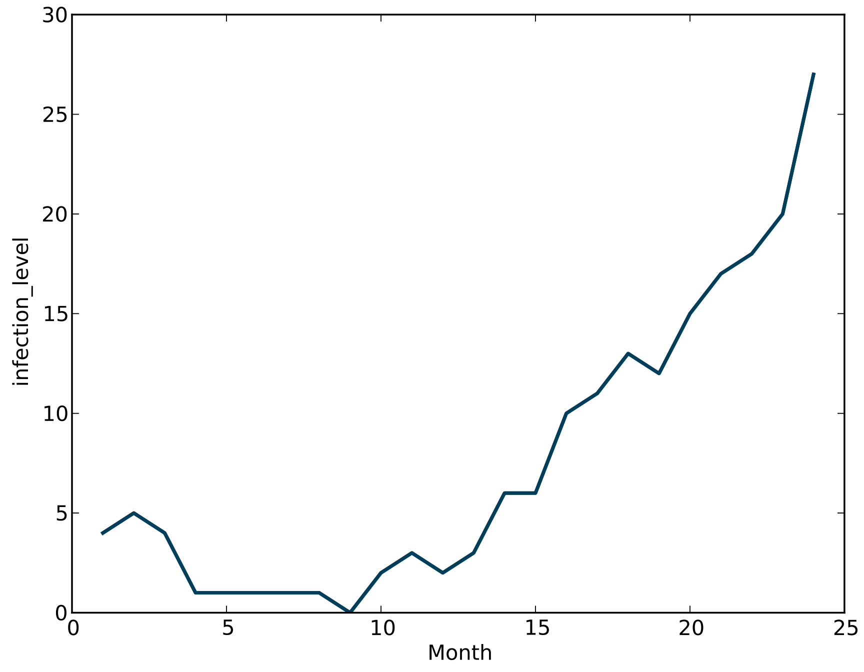


(b) Number of infected squirrels in the landscape for each month

Figure 4: Squirrels metrics



(a) Population influx for all of the cells in the landscape for each month



(b) Infection level for all of the cells in the landscape for each month

Figure 5: Landscape metrics

6 Testing

In this section, the testing mechanisms of the simulation are presented.

6.1 Functional Testing

Each functional unit of the framework and most of the simulation is responsible for checking the validity of the inputs. Any unexpected value leads the printing of an appropriate message to the standard error output and the finalisation of the simulation. The proper usage of the framework as explained before points to the zero failures in the framework.

6.2 Behavioural Testing

There is not an established testing mechanism for the simulation. Nevertheless, a few scenarios have been created that may be used as indicators for a correct implementation of the simulation. The scenarios are run using **test.sh** script. These scenarios and their expected outputs are:

1. - **Scenario** There are not infected squirrels
   - **Output** No one gets infected or dies, the population is constantly increasing
2. - **Scenario** There are not infected squirrels and there 170 squirrels initially spawned
   - **Output** The same as scenario 1 and the simulation will probably finish in an early stage
3. - **Scenario** There are 5 squirrels and all of them are infected
   - **Output** All of the squirrels will die and the simulation will probably finish in an early stage
4. - **Scenario** Running on 1 process
   - **Output** The simulation will terminate it need 2 or more processes
5. - **Scenario** Set 48 maximum months
   - **Output** The simulation will run for 48 months
6. - **Scenario** Running on 36 process, 7 squirrels
   - **Output** A few worker may never been assigned with actors

In each occasion, in the end of the simulation the sum of the number of spawned and died squirrels in each worker must equal to the number of spawned and died squirrels according to master's report.

6.3 Observation based

Scientists run many times the simulation and using the average values they get the final results. Simulating with the same results, we can denote that the results are similar if you compare e.g. Figure 5 (a) with 6.

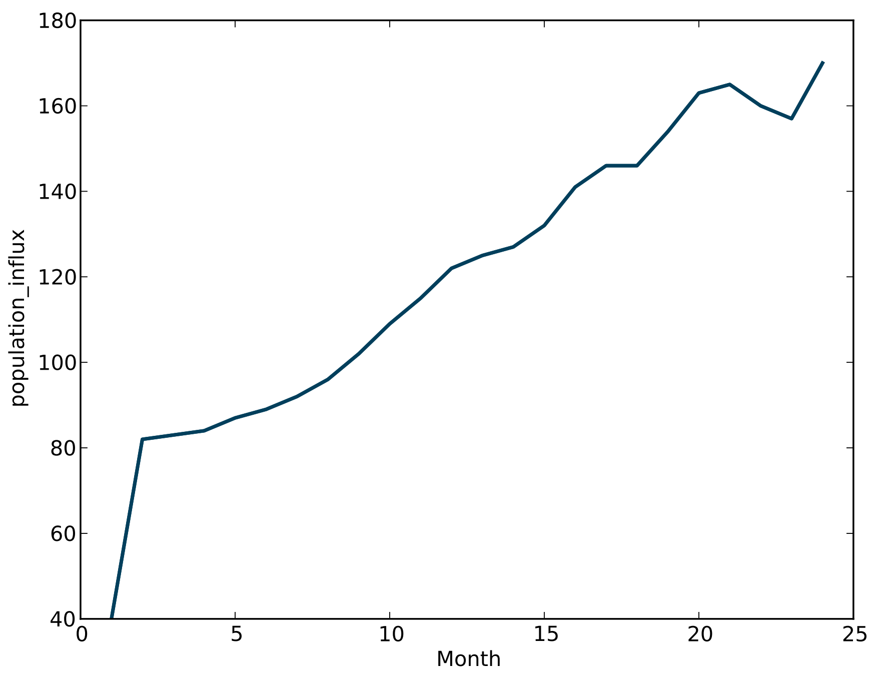


Figure 6: Population influx for all of the cells in the landscape for each month, experiment 2

7 Further work

The current implementation of the actor framework could be expanded to provide mechanisms such as load balancing or multithreaded execution inside each unit of execution. A well-documented implementation, used on a variety of simulations could improve the framework in order to be a reliable platform for the development of a project that can exploit the actor parallel design pattern.