|epcc|

Parallel Design Patterns
Assignment 2

B087928

May 9, 2016

1 Introduction

There are many parallel design patterns. Some are simpler than others to implement and more common. All have different advantages and disadvantages which determines which situation they are best suited to. Choice of pattern is therefore problem dependent.

We report on the design of an actor pattern code used to parallelise a biologist‘s simulation of the birth and death of red squirrels and the spread of squirrelpox in an environment. We do not provide the specific details of the biologist's model here, just a description of the design of the code as well as some comments on performance. We conclude by showing some results obtained from running the program.

Our code design description shall split into sections based on all the different actors involved in the program plus the master process which each in turn correspond to a module of the code. This is not entirely the case for the master process, which is in main.c however we shall describe all of main.c in this section. Each of these subsections shall have an associated MPI rank on which they always operate.

It may seem unusual to approach the design description in this way as each process communicates with other processes as so none of them can be considered completely independently. However, following the workflow is even less desirable since there is many different actions happening simultaneously.

There is a master process (rank 0) and a squirrel_master (rank 1). The squirrel_master keeps track of the squirrels in the status of the squirrels in the environment and is often referred to as the master_actor as it is in masteractor.c.

2 Code Design

There are 4 different types of actors: masteractor; clockactor; squirrelactor and cellactor as well as the master process.

General actor Structure Each actor in the program, with the exception of the clock actor, has a similar overall actor structure. Each enters a loop of the form:

```
int workerStatus = 1;
while(workerStatus){
   ...
   workerStatus=workerSleep();
}
```

We shall refer to this as the **workerStatus Loop**. Actors spend almost all their time within this loop, when they reach workerSleep they will return to the process pool and only continue when the master process sends for a new worker from the process pool or when shutDownPool call has been issued which is when the simulation is to complete. They each have a loop within this of the form:

```
while(1){
   ...
   if(terminate) break;
}
```

We shall refer to this as the **Living Loop**. The cellactor and masteractor only leave this loop when the simulation is complete and hence are never ‘woken back up‘ from workerSleep. However when a squirrel dies it sets terminate = 1 and re enters the process pool and may be woken back up when a new squirrel is born which shall be discussed further in section 2.5.

Message Receiving All sends in the program are non-blocking so we must consider how each process handles receives as these must be blocking inorder to ensure we receive them. When a non-clock actor is waiting to receive an instruction from the master or another actor the process loops in a do while loop. In this loop it probes for each potential message it could receive and calls the shouldWorkerStop function. If a probe comes back true then the actor receives the incoming message and breaks out of the do while loop and executes the function corresponding to the message received. If instead the shouldWorkerStop function returns that a shutDownPool has been issued then the do while loop also breaks with the value terminate = 1 and the worker sleeps. We shall refer to this do while loop as the **Receive Loop**.

2.1 Master Process (rank 0)

The master process is contained in main.c. All processes begin at the top of main.c. They each receive a different random seed value and block in a process pool when they reach:

```
int statusCode = processPoolInit();
```

from here rank 0 receives a statusCode of 2 and enters the masterprocess section of the code. It starts one worker process, this will be the masteractor process. It then enters a while loop where it calls masterPoll.

```
masterStatus = masterPoll();
```

Here it waits until it receives a message from an actor, at which point it will carry out the instruction received, loop and wait for the next instruction. When an actor calls shutdownPool the master process shall break out of the while loop and wait wait at processPoolFinalise until all actors are sleeping and finally the program will end.

2.2 Master Actor (rank 1)

The masteractor is the first to be woken up from processPoolInit and from there enters the squirrel_master function. The master wakes up workers for the 16 cells, the clockactor plus the inputted number of healthy squirrels followed by the inputted number of infected squirrels. Squirrels are initialised with a startWorkerProcess followed by sending the squirrels initial positions to the rank of the newly woken up worker by a call to the InitialiseSquirrel function.

Now that all initial actors (apart from those that will start when new squirrels are born) have been intialised we enter the while loops discussed at the beginning of section 2. The function of the masteractor is now to keep track of the status of the squirrels and shutDownPool when it detects that there are too many squirrels or that all of them have died. Furthermore, the masteractor will print to screen when it receives a message from the clock that a month has passed.

Receive Loop The options within the receive loop are:

```
Recv squirrel_status from squirrelactor          (1)
Recv message month_ended from clockactor         (2)
shouldWorkerStop                                 (3)
```

(1) receives an integer of -1, 0 or 1 if the squirrel dies; becomes infected or gives birth respectively. When one of these messages is received the receive loop is broken and the appropriate change of values, or in the case of a 1 a new worker thread is initialised as discussed at the beginning of this section. The code then returns to the receive loop and waits for another instruction.

(2) receives a 1 from the clockactor when a month has ended and then breaks out of the receive loop and prints the current squirrel values.

(3) receives a 1 if shutDownPool has been issued and then breaks out of receive loop and sets terminate = True.

2.3 Clock Actor (rank 18)

We may set month_time at the top of the clock actor. This variable represents the number of microseconds that the clock waits as the month_time. The clockactor shall most likely spend most of its time waiting at the usleep function for the new month to start. Then it sends to the masteractor and all the cells to print out their relevant values.

The clockactor sleeps if a shutDownPool has been called on a different actor, it checks this every month. If the simulation reaches 24 months the clock shall call shutDownPool itself and then sleep.

2.4 Cell Actor ranks (2-17)

The cellactor declares two small arrays to contain the infection level and population influx over the last 2 and 3 months respectively. Which array element to increment when a infection level or population influx is changed is determined from the month: infection_level[month%2] and population_in_flux[month%3]. See figure 1.

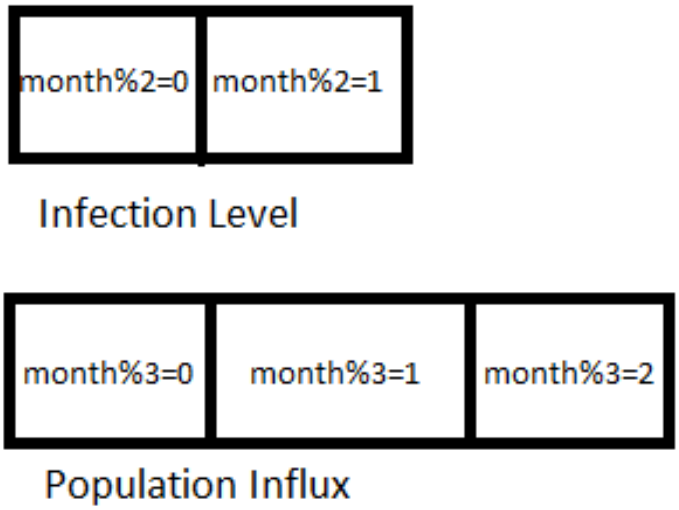


Figure 1: Arrays containing the infection level and population influx for the last 2 and 3 months respectively.

The level of infection, or the population influx for the last 2 or 3 months can then be easily calculated by summing up the elements of the respective arrays. This is done when the cell needs to send these values to back to a squirrel that has contacted it and when it is notified to print its values.

Receive Loop The options within the receive loop are:

```
Recv message month_ended from clockactor          (4)
Recv squirrel_status from squirrelactor           (5)
shouldWorkerStop                                  (6)
```

(2) receives a 1 from the clockactor when a month has ended and then breaks out of the receive loop and prints its infection level and population influx. The code then returns to the receive loop and waits for another instruction.

(1) receives an integer of 0 or 1 depending on whether an infected or healthy squirrel, respectively, has step on it (sent it the message). The receive loop is broken and the population influx for that month is increased by one; as is the infection level in the case of a 1. These calculated values are returned to the squirrel using status.MPI_SOURCE. The code then returns to the receive loop and waits for another instruction.

(3) receives a 1 if shutDownPool has been issued and then breaks out of receive loop and sets terminate = True.

2.5 Squirrel Actor (rank 19+)

Similarly to in the cellactor, the squirrel actor declares two arrays for infection level and population in flux. These arrays are of size 50 and are accessed by the value of stepnumber: infection_level[stepnumber] and population_in_flux[stepnumber]. Stepnumber is incremented, and stepnumber%50, calculated at the end of each of iteration of the **living loop**. I

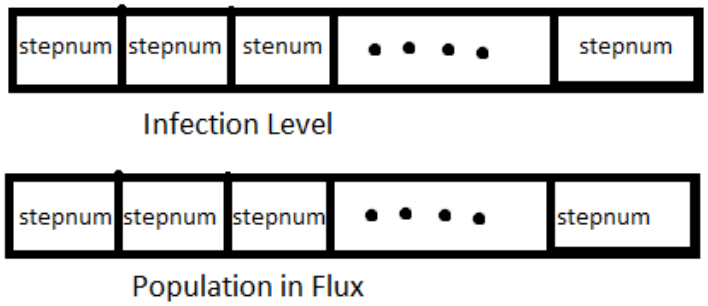


Figure 2: Arrays containing the infection level and population influx for the last 50 steps.

We can then determine the infection level or population in flux for the last 50 steps by simply summing the elements to these arrays similar to the cell code.

The squirrel sends the first message We begin at the start of the living loop. We have seen that in the other actor codes we wait in the receive loop (which is always at the top of the living loop) for a message to come in. The clock actor is an exception who just sends out messages periodically when a month ends and only receives if a shutDownPool is issued. The squirrel actor takes the a step first and therefore sends the first message. A squirrel step is calculated using the squirrelStep function and the cell it setps onto is determined from getCellFromPosition. Now the squirrel sends its status: 0 for unhealthy, 1 for healthy, to the cellnumber it has stepped onto and the cell receives into receive message (5) and processes as described in section 2.4.

Now the squirrelactor moves into its receive loop where the it waits for one of the two following receives:

`Recv values infection level and population influx` (7)

`shouldWorkerStop` (8)

(7) receives the current value of infection level and population influx in the cell the squirrel is occupying and adds them to their respective arrays (as seen in figure 2). The average infection level and average population influx can they be calculated by summing up the elements of these arrays and dividing by 50. The squirrel now reaches four consecutive if statements dependent on these calculated values:

1. Will squirrel give birth (and is it on its 50th step)?
   Yes: Notify masteractor (see receive (1)).
2. Will the squirrel catch the infection?
   Yes: Set healthy to 0 and send message to notify the masteractor (see receive (1)).
3. Is squirrel infected?
   Yes: Add one to its death clock.
4. Is death clock > 50?
   Yes: Will it die from infection? Yes: Notify masteractor (see receive (1)).

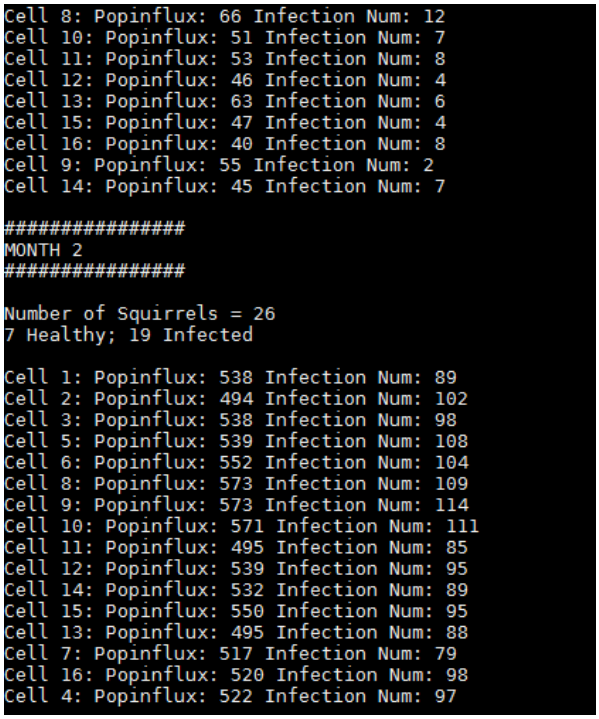
(8) receives a 1 if shutDownPool has been issued and then breaks out of receive loop and sets terminate = True. At which point the squirrelactor sleeps and the process rejoins the process pool. If it is re awaken then it will represent a new squirrel and it will re-initilaise its values within the workerStatus loop before re-entering the living loop.

3 Performance and Results

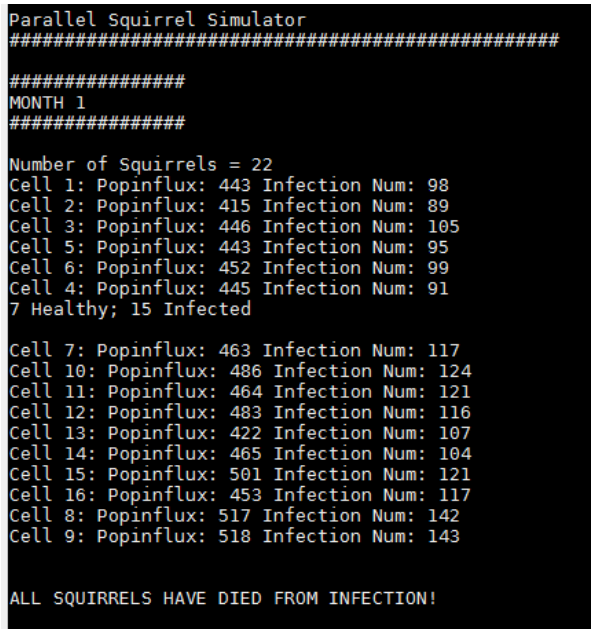
We ran on Morar with 64 processes as described in the accompanying readme document. The result of the simulation is stochastic. Each run is different to the last, however the result does have a very strong correlation with monthtime. If our monthtime is too long then the population either blows up or dies our extremely fast. For example we ran with

$$\texttt{month\_time} = 10000\mu s = 10ms$$

All squirrels died after just one month or two months.



(a) Run 1.



(b) Run 2.

Figure 3: Screenshots of two runs with month_time set to 10,000 μs = 10ms.

We found that we had much more interesting results for

$$\texttt{month\_time} = 1000\mu s = 1ms$$

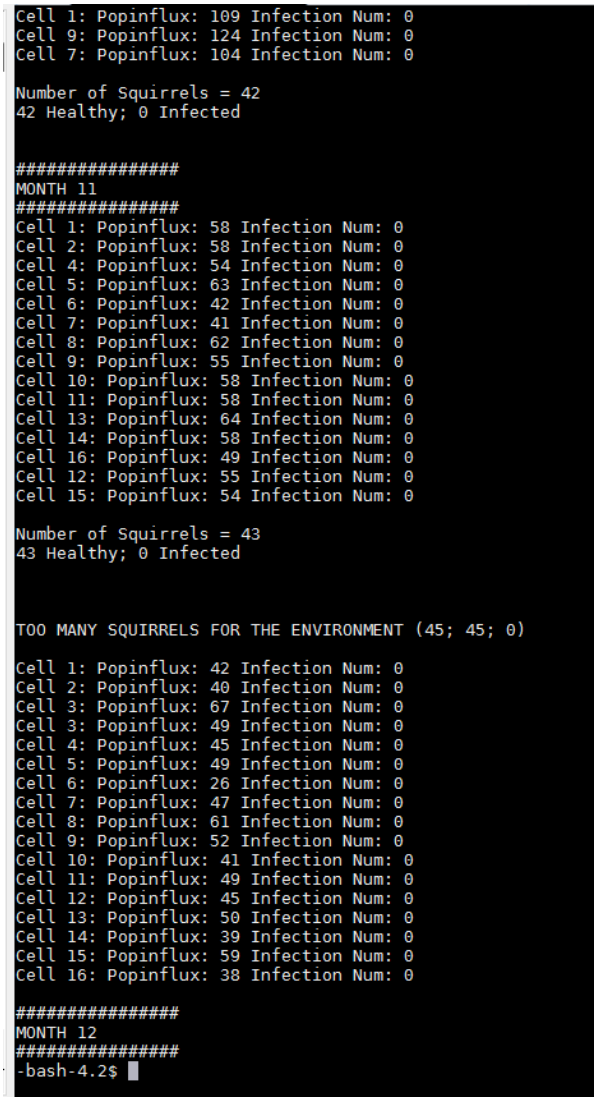
as can be seen in figure 4.

Shorter than this month_time however, destroys the output. Since we use non-blocking sends in the clock. Thus the clock sometimes finishes the next month before the cell responds. This is not necessarily considered a major issue since the cell receives the message from the clock after at most finishing with the squirrel it was having a communication with when the message was sent to it. So if the clock is getting so far ahead then it is also going at a speed which is simply too fast for anything interesting to happen in each month. Ie. if the output was somehow perfect there would still be a poor simulation since squirrels would make at most one or two steps per month.

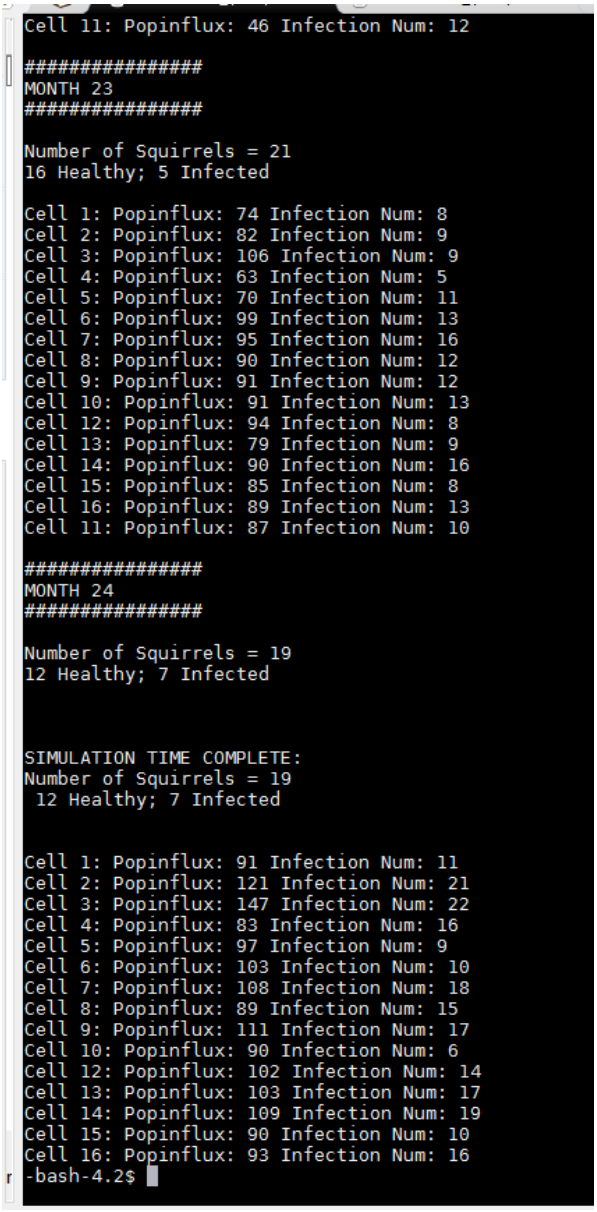
It is clear that synchronicity of our outputs depends on the rate of communication between cells and squirrels. Many communications per months will make our output neat and synchronised while one or less communications per month will result in very poor synchronicity and

output may correspond to wrong months. Luckily these more synchronous outputs also correspond to more sensible simulation parameters. This is demonstrated in figures 3 and 4. Thus the optimal simulation time is considered to be in the range

$$\texttt{month\_time} = 1000\mu s = 1ms \ \texttt{to} \ \texttt{month\_time} = 10000\mu s = 10ms$$



(a) Run 1.



(b) Run 2.

Figure 4: Screenshots of two runs with month_time set to 1,000 μs = 1ms.