# Parallel Design Patterns

## Coursework Part Two

## Introduction

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes Room 2.13

## Implement an Actor Pattern

- You may or may not have selected Actor Pattern as the solution to Part One
  - Don’t worry if you didn’t!

- You must **all** implement the Actor Pattern **even if this is not that pattern you selected in Part One**

- You must implement the pattern using the C, C++ or Fortran language. You must use MPI for parallelisation.
  - It must compile and run on Cirrus

*Due in 12 noon, Friday 5th April via Learn*

## Overview

- Coursework part two is worth 70% of the module mark
- Implementation – 75% of the coursework pt 2 mark (52.5% overall)
- Report – 25% of the coursework p2 mark (17.5% overall)
  - Detail the design of the implementation and how the pattern has been applied to the problem (such as what are your actors, how do they interact etc.)
  - If you have written a framework then also describe this and explain how it is to be used
  - Performance and scalability are secondary considerations, but credit will be given for a discussion about aspects of the code design that help or hinder performance.
  - Your report should include the output results from running your code
    - Credit will be given for some discussion around how you guarantee the correctness of your model

- Standard Solution:

  Correctly implementing the model using the Actor Pattern is enough to get you up to 69%

- Excellent Solution:

  For a distinction-level solution, I would expect to see good separation between code that implemented the actor pattern, and code that was specific to the biologists’ problem, ideally in the form of an “Actor Framework” that was then applied to the problem

## Provided code

- A process pool for creating/destroying actors
  - That you worked with in practical five
  - Also a test harness that further illustrates using this
  - In both C and Fortran
  - Optional for you to use this, but I think it might help with the dynamic creation/destruction of actors

- Some model functions that the biologists have supplied to you
  - Versions in C and Fortran

## Choice of Implementation

- You could implement using **one actor per process**
  - I believe that this is a more straightforward implementation
  - The process pool code as provided could be used as part of this implementation

- You could implement using **multiple actors per process**
  - This is, I think, significantly harder. Try at your own risk.
  - The process pool code is unlikely to help with this implementation

- I'd rather see a working implementation of the former than a broken implementation of the latter!
- Doing the first one very well, implemented using a clean, abstract framework, is sufficient for a distinction mark!

## General comments

- Well written, commented code will be rewarded
  - As will inclusion of a makefile, readme and a submission script
- There will be no extra marks for using source control or implementing unit tests etc
  - Although feel free to utilise these things if they help you write a correct implementation
- You might find practical 5 (mergesort using a process pool) and practical 4 (frameworks) especially useful
- The actor pattern promotes non-determinism, as does the problem & biologist functions themselves
  - This is fine, instead scientists would do multiple runs of your model and take the average of the results to get a final answer
  - There is no need to worry about exact reproducibility, either from run to run or machine to machine
  - Credit will be given for some discussion in your report about how you ensure correctness of your model

## A short introduction to the process pool code

- There is a process pool code provided
  - Both C (pool.c) and Fortran (pool.F90) versions provided
  - Treat this as a black box - unless you really want to, there should be no need to modify this code

- Practical five illustrated using this code
  - The sample solutions to this are online now

- Please do not let the “master/worker” terminology confuse you. In summary, the work assigned to each worker in this case is “be an actor, and continue to be an actor until you die, or until the program terminates”

## ProcessPool: Important Considerations (1)

- Your code must call **`MPI_Init`** before any of the following calls
- Every process must call **`processPoolInit`**

- The master process keeps track of which processes are active, but any process can call **`startWorkerProcess`** to request that a new worker is created/awoken. An ID is returned by this call which can be used to send a message to the new process.
  - The communications between the master and worker required to make this happen occur “behind the scenes”

## ProcessPool: Important Considerations

- All actors should be implemented using workers

```
while (workerStatus) {
   int parentId = getCommandData();

   // insert code here which implements being an actor

   workerStatus=workerSleep();
}
```

- The master process will probably do the job of creating the initial actors in the simulation
  - And then calls *masterpoll* repeatedly to poll for commands from workers

## ProcessPool: Important Considerations

- All actors **must** call **`shouldWorkerStop`** at regular intervals to allow the master to terminate the program if required. If **`shouldWorkerStop`** returns true, then it is **your** responsibility to ensure that the flow of control returns to the line after the call to **`workerCode`**

- `The process pool uses MPI tags 16384 & 16383`
  - `So avoid using these tags in your code`

## Biologist provided code

- void **initialiseRNG**(long* seed)
  - Call this once at the start of your program **on each process**. The value should be **negative**, **non-zero** and **different on every process**. You could seed with the value *-1-rank* where *rank* is the process's MPI rank. Out output, *seed*, becomes the random number generator's *state* that is passed to other functions.
- int **willGiveBirth**(float avg_pop, long* state)
  - Given the average population of the cells visited (i.e. the sum of their *populationInflux* values divided by the number of steps this has been taken over, 50), this returns true or false depicting whether the squirrel has given birth or not.
- int **willCatchDisease**(float avg_inf_level, long* state)
  - Given the average inflection level of the cells visited (i.e. the sum of their *infectionLevel* values divided by the number of steps that this has been taken over, 50), this returns true or false depicting whether the squirrel has caught the disease.

## Biologist provided code

- int **willDie**(long* state)
  - Returns true with a probability of 1 in 6. This is called for a squirrel to see if it is to die when it is infected (it should do 50 infected steps before calling this.)

- void **squirrelStep**(float x, float y, float* x_new, float* y_new, long* state)
  - Returns the position of a squirrel after it has stepped from position (*x*,*y*) to a new position, (*x_new, y_new*.) *x_new* and *y_new* can point to *x* and *y*. Each x_new and y_new is a value between 0.0 and 0.99

- int **getCellFromPosition**(float x, float y)
  - Given an (*x*,*y*) co-ordinate pair, this returns an integer representing the land cell that this corresponds to.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |