TÉCNICO LISBOA

Parallel and Distributed Computing

Project Assignment

FOXES AND RABBITS

.

ECOSYSTEM SIMULATOR

Version 1.7 (14/03/2022)

2021/2022

3rd Quarter

Contents

Revisions

Version 1.7 (March 14, 2022)	Minor change to conflict resolution rules
Version 1.6 (March 7, 2022)	Fixed deadline for serial version
Version 1.5 (March 3, 2022)	Minor change to world generation, updated sample output
Version 1.4 (February 13, 2022)	Random input format, sample output
Version 1.3 (February 13, 2022)	Changed to random input format, added sample output
Version 1.2 (January 27, 2022)	Swapped M & N, added count of animals
Version 1.1 (January 20, 2022)	Clarification of some parts of the text
Version 1.0 (December 29, 2021)	Initial version

1 Introduction

The purpose of this class project is to give students hands-on experience in parallel programming on both shared-memory and distributed-memory systems, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a program to simulate an ecosystem with two species: the Iberian fox (*Vulpes vulpes*) and the Iberian rabbit (*Oryctolagus cuniculus*).

The simulation takes place on a square grid containing cells. At the start, some of the cells are occupied by either a rabbit, a fox, or a rock, the rest are empty. The simulation consists of computing how the population evolves over discrete time steps (generations) according to certain rules described next.

2 Simulation Rules

Initially the grid is populated with rocks, rabbits and foxes. The world grid has a finite size and animals cannot move outside its boundaries. The i-axis (vertical) and j-axis (horizontal) both start at 0 (in the upper left corner of the grid) and end at limit-1 (with dimensions supplied on the command line for each simulation).

The animals move in this grid, and can breed and/or die. At the start of each new generation, the age of each animal is incremented. If its age reaches its breeding period, then it reproduces as described next. Foxes may starve if they don't eat a rabbit within a specified number of generations. The period for breeding and starving is defined at the start of the simulation. All animals are born with age 0, when the simulation is started and a new generation is going to be computed they have all been incremented so that during generation 1 all initial animals have age 1.

2.1 Rules for Rocks

Rocks don't move and neither animal can occupy cells with rocks (they are too steep to climb).

2.2 Rules for Rabbits

At each time step, a rabbit tries to move to a neighboring empty cell, picked according to the method described below if there are multiple empty neighbors. If no neighboring cell is empty, it stays. Rabbits move up or down, left or right, but not diagonally (like Rooks not Bishops).

If the age of the rabbit reaches the breeding age, when it moves it breeds, leaving behind a rabbit of age 0, and its age is reset to 0. A rabbit cannot breed if it doesn't move (and its age doesn't get reset until it actually breeds).

Rabbits never starve.

2.3 Rules for Foxes

At each time step, if one of the neighboring cells has a rabbit, the fox moves to that cell eating the rabbit. If multiple neighboring cells have rabbits, then one is chosen using the method described below. If no neighbors have rabbits and if one of the neighboring cells is empty, the fox moves there (picked using the method described below from empty neighbors if there is more than one). Otherwise, it stays. Same as rabbits, foxes move up or down, left or right, but not diagonally.

If a fox reaches a breeding age, when it moves it breeds, leaving behind a fox of age 0 (both for breeding and eating), and its breeding age is reset to 0. A fox cannot breed if it doesn't move (and its breeding age doesn't get reset until it actually breeds).

Foxes only eat rabbits, not other foxes. If a fox reaches a starvation age (generations since last having eaten) and doesn't eat in the current generation it dies. Hence, death occurs at the end of the current generation, namely after having reproduced.

2.4 Traversal Order Independence

Since we want the simulation to have the same result independently of the order that the individual cells are processed, you should implement the simulator using a so-called *red-black scheme* (as in a checkerboard, depicted in Figure 1) for updating the board for each generation.

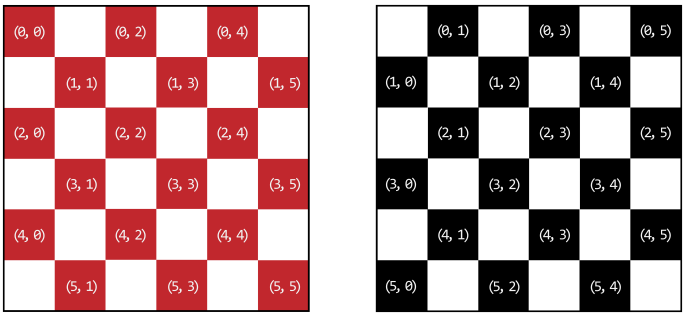

Figure 1: Red-black cell organization.

This means that a actually generation consists of two sub-generations. In the first sub-generation, only the red cells are processed, and in the second sub-generation the black cells are processed. In an even numbered row red cells are the ones with an even column number, and in an odd numbered row red cells are the ones with an odd column number.

The red-black scheme will help parallelize the computation of the cells. Cells of the same color are diagonal to each other, while the animals can only move parallel to the axes. This allows the evaluation of all the positions in a sub-generation, *i.e.*, over the red or black cells, in parallel. The red-black scheme allows you to think of each sub-generation as a separate parallel loop over the red (or black) cells, with no dependencies between the iterations of the loop.

Note that in the red-black scheme a rabbit or fox could end up moving twice in a generation. You need to prevent this, making sure that if the animal has moved in first sub-generation it will not move again in the next.

The rules that follow about selecting cells and resolving conflicts apply for each sub-generation.

2.5 Rules for Selecting a Cell when Multiple Choices Are Possible

If multiple cells are open for either a rabbit or a fox to move to, or for a fox to move to eat a rabbit, number the possible choices starting from 0, clockwise starting from the 12:00 position (*i.e.*, up, right, down, left). Note that only cells that are unoccupied (for moves) or occupied by rabbits (for foxes to eat) should be numbered. Let p be the number of possible moves.

Determine C, the grid cell number where the animal being evaluated is positioned. If the position of this cell is (i, j) in a world grid with $(0, 0)$ as the grid origin, and $M \times N$ the grid size, the grid cell number is given by $C = i \times N + j$.

Then, the cell to select is then determined by $C \mod p$. For example, if there are $p = 3$ possible cells to choose from, say up, down and left, then if $C \mod p$ is 0 the selected cell is up from the current cell; if it is 1 then select down; and if it is 2 then select left.

2.6 Resolving Conflicts

A cell may get updated multiple times during one generation due to rabbit or fox movements. If a cell is updated multiple times in a single generation, the conflict is resolved as follows:

1. If two rabbits end up in a cell, then the cell ends up with the rabbit with the greatest current age (closest to breeding – bigger rabbits win). The other rabbit disappears.
2. If two foxes end up in a cell, then, just like in the case of the rabbits, the cell ends up with the fox with the greatest current age (closest to breeding – bigger foxes win). If the foxes have the same age, the resulting fox gets the lowest starvation age of the tied foxes. The other fox disappears.
3. If a fox and a rabbit end up in a cell, of course the fox eats the rabbit and gets its starvation age reset.

Multiple conflicts can be solved applying the above rules in any order.

3 Implementation Details

3.1 Input Data

Your program should take ten command line parameters, all positive integers:

```
$ foxes-rabbits <# generations> <M> <N> <# rocks> <# rabbits> \
    <rabbit breeding> <# foxes> <fox breeding> <fox starvation> <seed>
```

To ensure that the world is generated the same for everyone, please use the code sequence provided in Appendix A. Note that because of the way this routine works (discards elements that fall in the same position), the initial world may end up with less species than what is specified in the command line.

3.2 Output Data

Your program should send to the standard output, `stdout`, just three integers in one line separated with one space in this order: the final number of rocks, rabbits and foxes.

The submitted programs should only print this information (and **nothing else!**) to the standard output, so that we can automatically validate the result.

The project **cannot be graded** unless you follow strictly these input and output rules!

3.3 Measuring Execution Time

To make sure everyone uses the same measure for the execution time, the routine `omp_get_wtime()` from OpenMP will be used, which measures real time (also known as "wall-clock time"). This same routine should be used by all three versions of your project.

Hence, your programs should have a structure similar to this:

```
#include <omp.h>
<...>

int main(int argc, char *argv[])
{
    double exec_time;

    generate_world();
    exec_time = -omp_get_wtime();

    run_simulation();

    exec_time += omp_get_wtime();
    fprintf(stderr, "%.1fs\n", exec_time);

    print_result();       // to the stdout!
 }
```

In this way the execution time will be sent to the standard error, `stderr`. This separation allows, if needed, sending the standard output to `/dev/null`.

Because this time routine is part of OpenMP, you need the include `omp.h` and compile all your programs with the flag `-fopenmp`.

3.4 Sample Problem

We will have available several public instances for you to compare your results against. Naturally, the evaluation will be carried on a set of private instances.

```
$ foxes-rabbits 22 4 4 3 2 2 1 10 4 123
0.0s
3 0 2
```

To illustrate, this are what the first couple of generations look like:

```
Initial world:
---------------
   00|01|02|03|
00:  |  |  |  |
01:  | F|  | R|
02: R| *|  |  |
03:  | *|  | *|
---------------
After generation 1, red
---------------
   00|01|02|03|
00:  |  |  |  |
01: F|  |  |  |
02:  | *|  | R|
03:  | *|  | *|
---------------
After generation 1, black
---------------
```

```
   00|01|02|03|
00:  |  |  |  |
01: F|  |  |  |
02:  | *|  | R|
03:  | *|  | *|
---------------
After generation 2, red
---------------
   00|01|02|03|
00:  |  |  |  |
01: F|  |  |  |
02:  | *|  | R|
03:  | *|  | *|
---------------
After generation 2, black
---------------
   00|01|02|03|
00:  |  |  |  |
01:  | F|  |  |
02:  | *| R| R|
03:  | *|  | *|
---------------
```

4 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C (or C++). Name the source file of this implementation `foxes-rabbits.c`. As stated, your program should expect five input parameters.

Make sure to include a Makefile so that the simple command

```
$ make
```

should generate an executable with the same name as the source file (minus the extension). Also, to be uniform across groups, in this makefile please use the `gcc` compiler with optimization flag `-O3`.

This version will serve as your base for comparisons and must be as efficient as possible.

5 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `foxes-rabbits-omp.c`. You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization as effective and as scalable as possible. Be careful about synchronization and load balancing!

Important note: in order to test for scalability, we will run this program assigning different values to the shell variable `OMP_NUM_THREADS`. Please do not override this value in your program, otherwise we will not be able to properly evaluate the scalability of your program.

6 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `foxes-rabbits-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to create independent tasks, taking into account the minimization of the impact of communication costs. You are encouraged to explore different approaches for the problem decomposition.

Note that this distributed version should permit running larger instances, namely instances that do not fit in the memory of a single machine.

Extra credits will be given to groups that present a combined MPI+OpenMP implementation.

7 What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (**please use the filenames indicated above**), and a table with the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks for both OpenMP and MPI, and additionally 16, 32 and 64 for MPI).

For both the OpenMP and MPI versions (not for serial), you must also submit a short report about the results (2-4 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why
- how was load balancing addressed
- what are the performance results, and are they what you expected

The code, makefile and report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>serial.zip`, `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

1st due date, serial version: **March 22nd**, until 23:59.

2nd due date (OpenMP): **April 1st**, until 23:59.

3rd due date (MPI): **April 22nd**, until 23:59.

A Code for World Creation

```
// Returns a uniformly distributed random number in the interval [0,1[
float r4_uni(uint32_t *seed)
{
    int seed_input, sseed;

    seed_input = *seed;
    *seed ^= (*seed << 13);
    *seed ^= (*seed >> 17);
    *seed ^= (*seed << 5);
    sseed = *seed;
    return 0.5 + 0.2328306e-09 * (seed_input + sseed);
}

void generate_element(int n, char atype, uint32_t *seed)
{
    int i, j, k;

    for(k = 0; k < n; k++) {
        i = M * r4_uni(seed);
        j = N * r4_uni(seed);
        if(position_empty(i, j))
            insert_element(i, j, atype);
    }
}

main()
{
...
    seed = atoi(argv[10]);
    generate_element(nrock, ROCK, &seed);
    generate_element(nrab, RABBIT, &seed);
    generate_element(nfox, FOX, &seed);
...
}
```