Parallel and Distributed Computing
Class Project

Squirrels & Wolves

DEAEngCmp , EMDC, MEEC, MEIC
2013/2014

The purpose of this class project is to gain experience in parallel programming on an SMP and a multicomputer, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a program that simulates an ecosystem with two species: the wolf (*Canis Lupus*) and the squirrel (*Sciurus vulgaris*).

The simulation takes place on a square grid containing cells. At the start, some of the cells are occupied, the rest are empty. The simulation consists of constructing successive generations of the grid. To understand how the simulation works, imagine the world is divided into a square 2D-grid, where:

- each grid cell can be empty or have a wolf, a squirrel, a tree or ice.
- the grid is initially populated with squirrels, wolves, ice or trees in a pre-defined manner, and
- the population evolves over discrete time steps (generations) according to certain rules.

Rules for Squirrels

- At each time step, a squirrel tries to move to a neighboring cell, picking according to the method described below if there are multiple empty neighbors. Squirrels move up or down, left or right, but not diagonally (like Rooks not Bishops).
- If a squirrel completes a breeding period, when it moves it breeds, leaving behind a squirrel at the beginning of the breeding period, and also starts a new breeding period. A squirrel cannot breed if it doesn't move (and its breeding period doesn't restart until it actually breeds).
- Squirrels never starve.

Rules for Wolves

- At each time step, if one of the neighboring cells has a squirrel, the wolf moves to that cell eating the squirrel and increases its current starvation period. If multiple neighboring cells have squirrels, then one is chosen using the method described below. If no neighbors have squirrels and if one of the neighboring cells is empty, the wolf moves there (picking using the method described below from empty neighbors if there is more than one). Otherwise, it stays. Wolves move up or down, left or right, but not diagonally (like Rooks not Bishops).
- If a wolf completes a breeding period, when it moves it breeds, leaving behind a wolf at the beginning of breeding and starvation periods. A wolf cannot breed if it doesn't move (and its breeding period doesn't restart until it actually breeds).
- Wolves only eat squirrels, not other wolves. If a wolf exhausts a starvation period (time steps since it last ate a squirrel), it dies.

More on Rules for Squirrels and Wolves

The specification above is not precise about when starving or breeding periods are restarted, and whether starving or breeding takes precedence. The right way to think about it is that an animal's age is incremented between generations (and a generation consists of a red and a black sub-generation, as described in the next section). So if a wolf completes its starvation period in a generation, it dies, since the starvation period did not get restarted by the end of the previous generation. As for breeding, an animal is eligible to breed in the generation after its current breeding period reaches the animal's minimum breeding period (supplied on the command line).

Rules for Ice

Ice doesn't move and neither animal can occupy cells with ice (they are too slippery to walk on).

Rules for Trees

Trees don't move and only squirrels can share a cell with a tree (they are too steep for wolves to climb).

Traversal Order Independence

Since we want the order in which the individual cells are processed to not matter in how the simulation evolves, you should implement the simulator using a so-called red-black scheme (as in a checkerboard) for updating the board for each generation. That means that a generation consists of 2 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 allows you to think of each sub-generation as a separate parallel (forall) loop over the red (or black) cells, with no dependences between the iterations of the loop. Note that in the red-black scheme a squirrel or wolf may end up moving twice in a generation. The rules that follow about selecting cells and resolving conflicts apply for each sub-generation.

Rules for Selecting a Cell when Multiple Choices Are Possible

If multiple cells are possible movement destinations for either a squirrel or a wolf:

- 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 squirrels (for wolves to eat), should be numbered. Call the number of possible cells p.
- Compute the grid cell number of the cell being evaluated. If the cell is at position (i,j) in the world grid with (0,0) the grid origin, and the grid is of size MxN, the grid cell number is $C = i \times N + j$.

- The cell to select is then determined by C mod p. For example, if there are 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.

Conflicts

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

- If 2 or more wolves end up in a cell, then the cell ends up with the wolf with the smallest current starvation period (farthest from starvation – less hungry wolves win), and the resulting wolf gets the breeding period of the wolf that had the smallest current starvation period. If 2 or more wolves have the same smallest starvation period, the resulting wolf gets the greatest breeding period of the tied wolves. The other wolf(ves) disappear.
- If 2 or more squirrels end up in a cell, apply the same rule as for wolves.
- If a wolf and one or more squirrel(s) end up in a cell, then the cell ends up with a wolf - the wolf eats the squirrel and adds to its starvation clock the starvation period times the number of squirrels it ate. Any other squirrel(s) and/or wolf(ves) disappear.

For this project the world grid has finite size. The x-axis and y-axis both start at 0 (in the upper left corner of the grid) and end at a limit.

Part 1: Serial

Write a serial implementation of the above program in C. Name the source file `wolves-squirrels-serial.c`.

Input

- Size of the side of the square grid (first line of the input file)
- Distribution of wolves (w), squirrels (s), ice (i) and trees (t) in the following file format (x-coordinate, y-coordinate, entity)

```
1 3 s
2 2 i
2 5 w
4 4 t
...
```

- Wolf breeding period
- Squirrel breeding period
- Wolf starvation period
- Number of generations (iterations)

Your program should take five command line arguments: the name of the data file, the wolf and squirrel breeding periods, the wolf starvation period, and the number of generations to iterate.

To be more specific, the command line of your program (e.g., for a sequential version) should be:

```
wolves-squirrels-serial <input file name> <wolf breeding period> <squirrel
breeding period> <wolf starvation period> <# of generations>
```

Output

The output of your program should be the configuration of the grid at the end of the given number of generations. Use the same format as specified for the input file (including the size in the first line). The output should be ordered, by lines than columns (as in the example above). Use the symbol $ to represent squirrels on a tree.

Suggested Data Structures

A 2-D grid of cells

```
    struct world {
        int type /* Wolf, Squirrel, etc. */
        int breeding_period;
        int starvation_period;
    } world[MAX][MAX];
```

At a high level, the logic of the serial (non-parallel) program is:

- Initialize world array.
- In each time step, go through the items in the order in which they are stored and perform updates.

Part 2: OpenMP

Write an OpenMP implementation of the Wolves and Squirrels program as in the Part 1, with the same rules. Name this source code `wolves-squirrels-omp.c`. There are now some complications to consider:

- You need to distribute the 2D world grid across threads, in 1 or 2 dimensions. Each thread is responsible for a 2D grid of world cells.
- For communication, each thread needs data from up to 4 neighboring threads.
- Two challenges are load balancing and potential for conflicts.

Conflicts

- Border cells for a given thread (the ones just outside the cells a thread is responsible for) may change during updates due to squirrel or wolf movements.
- Border cells need to be synchronized properly. Hence the update step involves communication between threads, but only at the borders of the grid assigned to each thread.

Load Balancing

- The workload distribution changes over time.
- You might want to experiment with row-wise/column-wise block distributions, or 2D/checkerboard block distributions.

Part 3: MPI

Write an MPI implementation of the Wolves and Squirrels program as for OpenMP, and deal with the same problems. Name this source code `wolves-squirrels-mpi.c`.

What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (please use file names that make it obvious which files correspond to which version, as described above) and the times to run the parallel versions on input data to be given later (for 1, 2, 4 and 8 processes).

You also must submit a short report about the results (1-2 pages) that explains:

- what decomposition was used
- how was load balancing done
- what are the performance results, and are they what you expected

You will turn in the serial version and OpenMP parallel version at the first due date, with the short report, and then the serial version again (hopefully the same) and the MPI parallel version at the second due date, with an updated report. Both the code and the report will be uploaded to the Fenix system in a zip file. Name these zip file as g<n>omp.zip and g<n>mpi.zip, where <n> is your group number.

1st due data (serial + OMP): **November 1st**
2nd due data (serial + MPI): **December 6th**