Grupo 35 OpenMP Report

1. Approach

Overview

The goal for this project is to create a simulation where several Foxes and Rabbits keep on moving in a 2D world.

The model follows some rules to determine the movement chosen by each animal and, for each particular movement, checks if the animal breeds or has to dispute the chosen position with another animal. Foxes always win the position in dispute with the rabbits, but they can also starve if several generations pass without eating any rabbit.

Each generation is an iteration of our program, which is divided in two sub-generations following a red-black scheme.

Beyond rabbits and foxes, our world is also populated with some rocks, which are always static and prevent any animal from occupying that position.

At the end of our program we output the number of foxes, rabbits and rocks, reflecting the final composition of our world.

Serial

The serial implementation is pretty straightforward. After initialising the world, we start our time loop, where each iteration is a generation and corresponds to two sub-generations (red + black).

Following the red-black scheme, we first process the red generations and afterwards the black, following these operations:

1. For each position of the world, check if there is an animal;
2. If there is an animal, increment the breeding age and check if there any available positions to move;
3. If there are, the animal chooses one;
4. As we move the animal, we check if the animal breeds - if so, leave a child behind (in the previous position) and reset his breeding age;
5. For the new position, we check if there any conflicts (other animals in that position);
6. Following the project rules, we check if the animal wins or loses the fight and update our world accordingly;

These operations are equally used in our red and black generations but we have taken two major concerns in our implementation:

1. From the red generation to the black one, we update a backup of our world to guaranteed that all animals have the attributes updated in the same generation and made decision based on the same information, ensuring that the decisions are independent from the order of the world processing in each sub-generation;
2. If some animal moved to a black position in the red subgeneration, that position of the world is invalid in the black subgeneration. This happens to prevent the animal from moving twice in the same generation;

OpenMP

Since OpenMP is based on Shared Memory Parallelism, we used the serial implementation as the base program for the parallel version. Since there were some problems due to memory management as some data was being shared through the entire program, we analysed our serial code to find which parts can be parallelized without changing the final result and how the tasks can be split as evenly as possible between threads.

As we identify the possible conflicts, we applied some changes to our serial implementation. Some global variables of our program were redefined as local variables - for instance the vector that stores the available moves for each animal - since it is very important that every thread keeps his own copy, which, looking backwards, could be also guaranteed using the "private" directive.

Furthermore, we also refactor our generations, creating a run_back and a run_red functions for each one. They basically do the same operations as the serial version but allow an easier interpretation of our program.

During the development of our project we went through several different strategies to try and minimise our runtime. We started with a solution that treated odd and even rows separately, the objective of this version was to minimise conflicts since most conflicts occur between consecutive rows. However even with this division, we later noticed that conflicts are still possible if two even rows decided to write in the odd row between them and vice-versa.

To avoid these conflicts we created a system of locks for each cell and whenever an animal moved up or down it would lock the cell where it was writing. Even though this initial solution was good it didn’t resolve all conflicts (due to up and down movements) and it didn’t maximise parallelism (due to odd and even rows running separately).

Based on the problems of the first version we decided to follow two new promising solutions, one that minimised conflicts and the other that maximised parallelism:

The first solution split the board in pairs of two lines and it ran adjacent pairs of lines separately. This is, first it runs the pairs of lines (1,2),(5,6),(9,10)... and then it runs the rest of the pairs of lines. The logic of this approach is to use one thread for each pair of lines avoiding any need for synchronisation between threads since there are no possible conflicts between them. This approach traded the need to resolve conflicts and synchronisation between threads for limited parallelism since only half the lines can be being processed at the same time.

The second solution used a *parallel for* that would run all the lines, and used the previously mentioned system of locks for each cell to lock the individual cells where each animal was writing. To avoid setting unnecessary locks and having too much synchronisation overhead, we use static scheduling and calculate the lines where possible conflicts between threads may occur and only activate the locks in those situations. This second solution ended up being slightly better than the first one (using pairs of lines) so we kept it as our final solution and will therefore be described in more detail further ahead.

2. Parallel Decomposition

As briefly mentioned before, our final solution uses a *parallel for* for each of the sub-generations(red/black), where each thread will run M/T lines, with M being the total number of lines and T being the total number of threads. To solve conflicts between two threads trying to write to the same cell position we have a set of M x N locks for each of the cells of the grid and whenever an animal wants to write in a certain cell it must take the lock for that cell or wait for it, in case it is already being used. When an animal finishes using a cell, it releases the corresponding lock.

Besides the two main *parallel fors*(one for each generation), there are still a couple other areas where we added some parallelization. At the end of each generation we needed to kill all the foxes that went past their starving age. For this we used a simple *parallel for* with *collapse(2)* since killing each fox can be done independently and without any conflicts. At the end of our program we also need to add all the foxes, rabbits and rocks that are still alive, and in this function we also used a *parallel for* with a *collapse(2)* since each cell can be processed independently and with a *reduction* + to optimise the speed of adding all the values together.

3. Synchronisation Concerns

As described in the approach section we had different synchronisation concerns throughout the project however for our last version the most important synchronisation concern is related with the usage of the locks. Locking and unlocking a cell every time an animal moves to a new position would lead to a lot of wasted time in synchronisation, since most writes do not interfere with other threads. To avoid unnecessary locks we calculate the lines of the matrix that each

thread will use and we only use the locks in lines that are in the border between two different threads.

4. Load Balancing

In terms of loading our solution is quite simple. Since we need to know exactly what group of lines each thread would run, we had to use static scheduling as our load balancing choice.

5. Performance Results

The next table as the results ran in Lab PCs for the following tests:

Test 1 - Input data: 300 6000 900 8000 200000 7 100000 12 20 9999
Test 2 - Input data: 4000 900 2000 100000 1000000 10 400000 30 30 12345
Test 3 - Input data: 100000 200 500 500 1000 3 600 6 10 1234
Test 4 - Input data : 20000 1000 800 100000 80000 10 1000 30 8 500

	Serial	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1	48,35	51,4	31,0	16,0	14,8	14,5
2	192,53	201,0	124,9	66,6	60,0	60,0
3	190,25	205,4	124,2	59,2	58,8	59,8
4	574,4	649,8	386,9	173,7	162,9	162,9

We are only showing the results for the bigger tests because all the small tests had times of smaller than 1 second and were, for that reason, useless for the comparison.

We have also organised some graphs with the information on the table and the speed up, so that the visualisation of the information is easier. The graphs are on the file *Information-graphs.pdf* which is in the delivered *zip* folder.

Observing the obtained results it is clear that the gain in performance is much more significant between 1 and 4 threads. With 8 and 16 threads there is still some performance increase but it is very small… This can be explained with the fact that the Lab computers only have 4 cores and therefore numbers of threads bigger than the number of cores will not be so efficient.