Grupo 35 MPI Report

1. Approach and Decomposition

In the beginning of the development of the project we decided to implement two solutions, one based on dividing the world matrix among the processes in a row-wise manner, and another that divided the world with a checkerboard decomposition. The decision to try to implement a block decomposition was based on the fact that, in theory, it should obtain better scalability when tested with a larger number of processes. This is based on its isoefficiency analysys(M(rowise) :C^2p / M(checkerboard) : C^2).

We weren't able to implement a block decomposition algorithm that worked well with all input possibilities (matrix dimensions and p processes) so we settled with a row wise decomposition **Note**: In the zip file we also sent the code for the incomplete implementation of the checkerboard algorithm.

Final version:

The final version of our project uses both MPI and OMP and divides the world matrix row wise. We tried to implement an approach that divided the world matrix in squared blocks, but unfortunately we did not succeed.

Our program will divide the matrix in blocks of *M/p* lines where *p* is the number of processes and M is the number of lines. If the number of processes exceeds the number of lines, the overflowing ones are removed from the “process world”. Inside the block of each process we divide it in groups of *B/t* lines*,* with *B* being the block size of the processor and *t* being the number of threads.

2. Communication

Each process is responsible for calculating a certain block of lines, however in order to guarantee the correctness of our program, each process will need to receive and use the two lines above and the two lines below its block.

In order to be aware of possible conflicts and movement of animals between blocks, each process needs to know what happens in the lines immediately above and below it, which I will call “critical lines”. However in order to calculate what happens in the “critical lines” around its region, each process also needs to be aware of what is in the lines above and below the “critical lines”, and that's why each process will need to send/receive both the two lines below and above it.

Each process will then communicate, at the end of each sub generation, with the process(es) above and/or below it using MPI_Irecv and MPI_Isend calls in order to send and receive the said lines. When all the generations are calculated, each process will then send its part of the matrix to the “root” process (rank 0), using a call to MPI_Gatherv that in turn will put all the parts together and calculate the final number of rocks, rabbits and foxes.

3. Synchronisation Concerns

The synchronisation concerns within each process were already discussed in the OMP report. From the MPI part of the code, most of the synchronisation concerns come from our use of non-blocking send and receive calls, however these concerns are easily solved by not touching the buffers used in MPI_Isend calls during communication and by using MPI_Wait calls before accessing the buffers used in the MPI_Irecv calls.

4. Load Balancing

We thought of two main ways of doing load balancing.

In the first method, the root process would keep sending individual lines to each process in a round robin sequence as they finish calculating the previous one. This way we might be able to achieve very good load balancing, however this method could not take advantage of OMP multithreading inside each block and would require a lot of communication overhead.

The second method instead of statically defining the size of the block for each process, it would dynamically change the size of the blocks at the end of each generation by slightly increasing the size of block for the fastest process and slightly reducing the size of the block for the slower process.

The second approach could take advantage of OMP inside each block and could fit well in our code but we didn’t have time to implement either approach.

5. Performance Results

We tried to run our program on the large instances for 2 , 4 , 8 and 16 processors in the RNL Cluster with 4 threads in each processor. Unfortunately for 32 and 64 processors our program gives wrong results sometimes, so we will not include those values. Sometimes test 4 will also give wrong values even for 4 , 8 and 16 processors, likely due to its very high number of generations increasing the likelihood of an error.

The inputs for each test are the following:

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

	2	4	8	16
1	36,9	34,6	31,8	24,4
2	156,2	155,6	120,7	157,8
3	435,6	393,5	373,8	329,9
4	162,4	194,4	220,2	212,9

In all but the last test, it is clear that the times are lower and lower the more processors we use, however they are still bigger than the times of the previous implementation, which only had OMP. This slower performance is likely due to the communication overhead, which for this problem does not justify the smaller amounts of computation in each processor. The high number of generations of the last test led to a very high number of communications and their corresponding overhead, causing the obtained times to follow an upwards irregular trend.