Foxes and Rabbits Project

MPI Implementation (3rd submission)

Parallel and Distributed Computing

Group	**11**
81120	Filipe Sousa
92513	Mafalda Ferreira
93114	Lucas Raimundo

MEEC/MEIC

IST-ALAMEDA

2021/2022

Contents

1 Parallelization Approach

We identified 4 sections with data parallelism in our code: red sub-generation; update world; black sub-generation; update world and kill foxes. These sections correspond to *for* cycles that iterate the lines and columns of the world.

Following *Foster's design methodology* for the sub-generations:

- **Partitioning**: Moving an animal.
- **Communication**: Send an animal to one of the neighboring cells.
- **Agglomeration**: Block of lines are processed together.
- **Mapping**: Distributed approach of block decomposition (slides 8).

Following *Foster's design methodology* for updating the world:

- **Partitioning**: Stabilizing a cell.
- **Communication**: No communication necessary.
- **Agglomeration**: Block of lines are processed together.
- **Mapping**: Distributed approach of block decomposition (slides 8).

By using the block decomposition code provided by the teachers, we are dividing the world by the processes in such a way that each process will only manipulate one contiguous block of the matrix throughout the entirety of the program execution. Due to this, in the beginning of the program, each process only generates its part of the world.

Updating the world is trivial. Since there are no dependencies between cells, each process can just process its own block with no need for communication. In the sub-generations, animals in the first line of the block, in order to choose where to jump correctly, need to know about the neighbors above, and need to be able to jump out of the block. The same applies to the last line and the line below.

To solve this, each process not only saves its block, but also the lines above and below it. We refer to these lines as *ghost lines*. This way, we can calculate the jump

correctly, and save the jumps that go out of the block. One important detail is that the processes don't process the animals in the ghost lines, that's the job of another process.

After a process finishes processing its block in a sub-generation, it needs to receive the neighboring processes' ghost lines (since animals could have jumped into the block), and it also sends them its ghost lines. To integrate the ghost line in the block, we apply the conflict resolution rules to the animals in the ghost line that have moved in this generation. Also, we only iterate the cells that are not the color of the current sub-generation since animals could have only jumped to that color. After integrating the ghost lines, a process sends the edge lines (not ghost lines) to the neighbors in order to keep updated ghost lines.

In order to calculate the result, each process counts the number of rabbits, foxes and rocks in its block, and we apply a reduce for each of those values with process 0 as the root. Then process 0 prints the result.

2 Load Balancing

The approach used considers static blocks distributed among the processes, where each process computes approximately the same amount of world positions. However, the number of animals differs for each block, resulting in a different amount of work.

In order to address the lack of load balancing, we created a second version of the project based on a processor farm model, which performs centralized dynamic load balancing.

The master process is responsible for sending a block of the world to each available slave process. The block contains a predefined chunk of lines to process and two ghost lines. Afterward, each slave computes the animal movements of the chunk, updates the entire block and returns it to the master process. The master process is responsible for replacing the world section with the new chunk of lines and solving conflicts between the received ghost lines and the original world lines.

Ideally, the processor farm model would be more efficient, since it would solve the lack of load balancing issue. Unfortunately, after running the code it resulted much worse time than the first version. This is probably due to the fact that this approach

requires heavy communication between the master and the slave processes. For each communication set, the processes need to send, in both directions, all memory positions associated with the block, resulting in higher latency.

In contrast, the static blocks approach provides lighter communication with only two ghost lines, which results in reduced communication time. Hence, the best option was to keep the solution based on static blocks.

As a side note, in the processor farm implementation the master process sends tasks to the slave processes and solves conflicts sequentially. This could have been improved by using threads, but due to lack of time, we chose to go with the status blocks version.

3 Results

Table 1 shows the execution times in seconds for 1, 2, 4, 8, 16, 32 and 64 processes, for each map.

Table 1: Execution time.

Instances/Processes	1	2	4	8	16	32	64
Large Instance 1*	13.8s	11.7s	10.0s	5.9s	3.7s	2.4s	2.1s
Large Instance 2*	81.1s	46.2s	38.2s	35.3s	29.3s	24.9s	25.4s
Large Instance 3*	227.7s	158,4s	80.3s	86.4s	83.5s	80.0s	81.8s
Large Instance 4*	55.1s	37.7s	21.2s	147.8s	199.6s	187.8s	334.4s

Looking at the obtained results, up to 4 processors, the execution times meet the expectations, but in the instances with more generations (Large instances 3 and 4), the results are quite odd and in the case of the Large Instance 4, the performance gets dramatically worse.

In the Large Instances 1 and 2, it's also worth noticing that when upgrading from 32 to 64 processes, the speedup is pretty much non existent.

In Table 2 we have the speedups for 2, 4, 8, 16, 32 and 64 processes. The maximum speedup we could theoretically get was the number of processes. Since we need

to send and receive the ghost lines to and from the other processes, we won't get a linear speedup because the execution time is limited by this communication between processes.

Table 2: Speedup over serial version.

Instances / Speedup	2	4	8	16	32	64
Large Instance 1*	1.2	1.4	2.3	3.7	5.8	6.6
Large Instance 2*	1.8	2.1	2.3	2.8	3.3	3.2
Large Instance 3*	1,4	2,8	2,6	2,7	2,8	2,8
Large Instance 4*	1.5	2.6	0.4	0.3	0.3	0.2

* Large Instance 1: r300-6000-900-8000-200000-7-100000-12-20-9999

* Large Instance 2: r4000-900-2000-100000-1000000-10-400000-30-30-12345

* Large Instance 3: r20000-1000-800-100000-80000-10-1000-30-8-500

* Large Instance 4: r100000-200-500-500-1000-3-600-6-10-1234

Looking at the speedups obtained, it can be said that the chosen approach is not very scalable. Generally speaking, depending on the number of generations, there is no benefit in performance from using more processes - the Large Instances 1 and 2 caps at 32 processes, and the Large Instance 3 caps at 4 processes.