INSTITUTO SUPERIOR TÉCNICO

Parallel and Distributed Computing

Foxes and Rabbits - MPI

Short Report

93088 – João Alexandre Ferreira Leitão
93106 – José Carlos Oliveira Brito
93202 – Xavier Batista Fernandes

Group 2

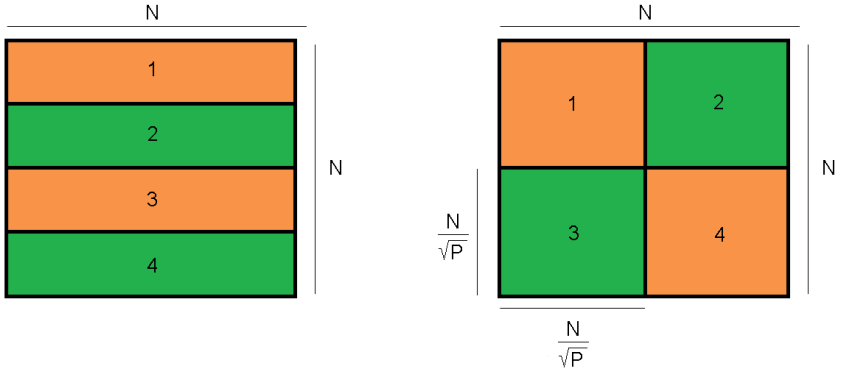
April 22, 2022

Approach

One downside of the OpenMP implementation of the project is that maps had to be small enough to fit on one machine. By creating a message passing version of the program we can utilize different machines, which increases the amount of available resources, allowing us to solve very large maps. Just like in the OpenMP version of the project, we once again attempt to exploit the existing parallelism within a generation. Previously, the world matrix was split into different tasks to be executed by different threads in the same processor. In this version of the project, the work is split across different processes running on different processors (and potentially different machines), with each process computing its part of the word in a serial manner. This approach brings a few challenges, especially when it comes to communication as, since the different processes don't share their memory address space, they must explicitly communicate with other processes they need data from, whereas previously communication between threads was implicit through shared memory. This communication is implemented through OpenMPI. Another challenge we had to tackle was the world generation logic, which must be changed in order to accommodate for maps that are too large to fit on one machine. To work around this issue, each process generates its own part of the world and, when a process requires some part of the world it did not generate, they request it from the process that did.

Decomposition

There are multiple ways of decomposing the world and distributing it to the different processes. It is possible to split the matrix into blocks that consist of rows of the world matrix, like we did in the OpenMP version of the project. This is a simple approach to the problem, as communication between processes consists of simply sending some rows to the processes above and below. There is also the possibility of splitting the matrix into blocks with a certain number of rows and columns, i.e., a checkerboard decomposition. This approach is more complex, as now processes will have to not only communicate rows to the processes above and below them, but also columns to the processes to the left and right and even some corner cells to processes diagonal to them. This is however, generally, a much more scalable approach, as, with the world split this way, the amount of communication processes have to make reduces as the number of processes increase. We present figure 1 to exemplify both decomposition approaches with an `NxN` square matrix and `P` processors.



(a) Row Decomposition (b) Checkerboard Decomposition

Figure 1 – Different decomposition techniques

As we can see, with the row wise decomposition, independently of the number of existing processes, the amount of data communicated by each process is always two rows of cells ($2\texttt{N} \times \texttt{sizeof(Cell)}$) (except for the top and bottom processes which communicate only one row of cells). However, with a checkerboard decomposition, the data communicated by each process depends on the total number of processes P, being equal to, at most, when a process has to communicate above it, below it and to the left and right (not shown on the figure), $4 \times \frac{\texttt{N}}{\sqrt{\texttt{P}}} \times \texttt{sizeof(Cell)}$.

All this being said, we begun by implementing a row decomposition, where we evenly divide the number of rows of the map across the different processes, using the following arithmetic.

$$\texttt{First Row} = \left\lfloor \texttt{id} \cdot \frac{\texttt{M}}{\texttt{P}} \right\rfloor \qquad\qquad \texttt{Last Row} = \left\lfloor (\texttt{id} + 1) \cdot \frac{\texttt{M}}{\texttt{P}} \right\rfloor - 1$$

where `id` corresponds to the process id, `M` corresponds to the total number of rows and `P` to the total number of processes. We then implemented a checkerboard decomposition, where we used `MPI_Dims_create` and `MPI_Cart_create`, to determine the grid utilized for this decomposition, i.e., how many processes would be on each axis. Knowing this, we could split rows and columns for each process with a slightly altered version of the above equations, taking into the account the number of processes and their respective ids in each axis.

Synchronization and Communication

In order to ensure program correctness, some synchronization and communication between processes is required. When trying to process the block's border cells, it is necessary to know the state of the adjacent cells, however, these cells are only present in other processes, meaning it is necessary for processes to be synchronized. To work around this problem, we made it so that when each processes allocated its part of the world, it also allocated space for ghost rows and columns that will hold the rows and columns adjacent to the border cells. At the start of each generation, the processes communicate the ghost rows to each other. We discovered that 3 ghost rows and columns were enough to avoid having to communicate every sub-generation, as we can simply process what happens in the communicated ghost cells during the red sub-generation to be able to correctly compute the black sub-generation. This way, we are able to communicate only at the start of each generation. We also attempted to increase the number of ghost rows and columns in order to be able to communicate less often while doing more computation, however, we did not notice any performance improvement, and so decided to stick with 3 ghost rows and columns. Communication between processes was done by first copying the required rows/columns to a buffer and then utilizing the `MPI_Irecv` and `MPI_Isend` functions to send that data to the right processes. `MPI_Waitall` is used to ensure that all messages have been sent and received before moving on to computation, this in turn also ensures processes are synchronized at the start of each generation. In the row-wise decomposition, communication only happened vertically, while for the checkerboard decomposition communication could be vertical, horizontal and diagonal (with our implementation, corner cells had to be communicated as well). We also attempted to make some computation overlap with communication, by attempting to process the non-border cells of each block (which don't depend on the ghost rows/columns) before the `MPI_Waitall` function, however, once again, there was no gain performance and so we decided to stick with what we had. Another piece of communication happens after computing all generations, where there

is a reduction of the final number of foxes, rabbits and rocks through the `MPI_Reduce` function.

Load Balancing

Unlike our OpenMP implementation, which creates large amounts of light-weight tasks and dynamically assigns them to the existing running threads (dynamic load balancing), our MPI implementation divides the world map in blocks, with the number of blocks being equal to the number of threads. This mapping is statically made, meaning before starting performing any computation on the cells, each process already knows which elements of the world map it is going to process. This work distribution contrasts with the one performed on the OpenMP version of the project, as each process in the MPI version has one contiguous block of data. The downfall of this approach is that, although all processes have approximately the same amount of cells to process, some cells may require more computation than others (empty cells vs cells with animals), depending on the seed generation of the world, and thus some processes may finish their work faster than others, remaining idle waiting for the process with the largest workload to end its computation. A more even load distribution could be achieved by assigning to each processor multiple but smaller blocks of the world map instead of a single and larger one. These blocks would be assigned to the cores in a interleaved way, mitigating the idle time of each processor as it becomes more likely that all cores process not only light-weighted blocks but also heavier-weighted ones in terms of workload.

Performance

Table 2 contains the execution time of different tests ran on the lab machines (excluding the lab2 machines which are faster) to benchmark the performance of our MPI implementation of the project (checkerboard decomposition). The table shows the execution times and speedups obtained for 8,16,32 and 64 processes (inside the parenthesis are the total number of machines), for the following testing maps: (Note that no serial time is provided for the final 3 tests as they are too big to fit on a single machine and run serially and so the shown speedup is the speedup relative to the 8 process execution time)

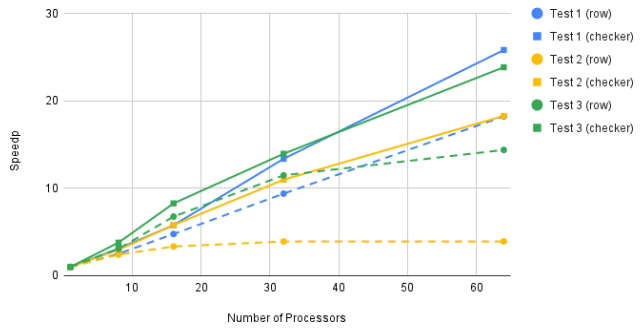
Table 1 – Executed tests

Test Number	#Gen	M	N	#Rocks	#Rabbits	Rabbit B.A.	#Foxes	Fox B.A.	Fox S.A.	Seed
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	50	30000	8000	1000000	120000000	4	30000000	7	10	101010
Test 5	20	15000	70000	1000000	300000000	3	50000000	5	10	101010
Test 6	20	80000	10000	1000000	120000000	3	20000000	5	10	101010

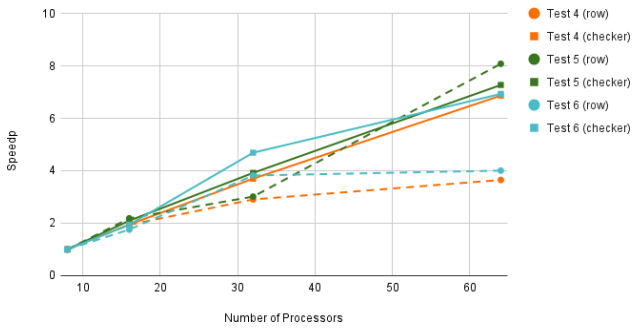
Table 2 – Execution times and speedups obtained for different tests and different numbers of processes (checkerboard decomposition)

	Test 1 [s]	Test 2 [s]	Test 3 [s]	Test 4 [s]	Test 5 [s]	Test 6 [s]
Serial	18.58	78.37	275.58	-	-	-
8 (2)	6.11 (3.04x)	26.49 (2.95x)	72.9 (3.77x)	51.62	119.29	75.94
16 (4)	3.22 (5.77x)	13.56 (5.77x)	33.34 (8.26x)	26.47 (1.95x)	56.91 (2.10x)	39.04 (1.94x)
32 (8)	1.62 (11.47x)	7.22 (10.85x)	22.62 (12.18x)	13.96 (3.96x)	30.44 (3.92x)	16.20 (4.69x)
64 (16)	0.85 (21.86x)	4.32 (18.14x)	11.76 (23.43x)	7.52 (6.86x)	16.41 (7.27x)	10.96 (6.92x)

Figure 2 shows a comparison of the obtained speedups for all tests and for 8,16,32 and 64 processors for the row-wise and checkerboard decompositions.



(a) Speedups relative to serial execution time - Tests 1, 2 and 3



(b) Speedups relative to 8 processor time - Tests 4, 5 and 6

Figure 2 – Performance comparison between Row-wise and Checkerboard decompositions

As we can see, in general, our checkerboard implementation scales very well with an increasing number of processes, as it increases by roughly 2 times as the number of processes doubles, unlike what is seen for the row-wise decomposition where the scaling isn't as good. This checks out with what we mentioned in the *Decomposition* section of the report. As the number of processes increases, message size decreases, and so, despite having a bigger number of total messages (in checkerboard decomposition, processes communicate horizontally, vertically and also diagonally), the checkerboard implementation ends up having a smaller communication overhead. One outlier was test 5, where performance was quite similar for both implementations, we suspect this could be due to workload imbalance. We also noticed some cases of super-linear speedup (for example, test 6, going from 16 to 32 processors) which can potentially be explained by the fact that increasing the number of processors decreases the chunk of the world each processor has to compute, meaning the key working set starts to fit in per-processor cache.

It is also to note that speedups were quite below their ideal values. There is a multitude of reasons as to why that could be the case. First we want to point out load balancing, in fact, our implementation does a completely static distribution of work and so if the map is unbalanced, execution time can suffer due to that. Another reason as to why speedups are below ideal could be due to the fact that the program is memory bound (data movement operations are predominant over computation), meaning performance is limited by available bandwidth resources and not by computational power.

One last thing we wish to mention is that it should be possible to improve the performance of our program even further by introducing OpenMP in our current implementation. If each process is computing its own part of the world on different processors, then, once communication is complete, we can take advantage of the available multithreading capabilities to speed up the computation of sub-generations and improve performance. We were capable of incorporating our previous OpenMP implementation into the MPI implementation, with each process splitting its own section of the world into tasks to be run by OpenMP threads (and making sure only one thread handled communication). Unfortunately, we were unable to test this hybrid implementation and compare the results with the pure MPI implementation due to the cluster being down.