Parallel and Distributed Computing Project: OpenMP

Vicente Silvestre 92730[1], Bernardo Silva 93365[1], Olivier Beck 104534[1]

[1]Group 29 - Instituto Superior Técnico

April 1, 2022

I Parallelization Approach

To parallelize the code, we started by looking at the initialization and end of generation routines. These were simple `for` loops with no dependencies or data races, so the solution for these was to apply an `omp for` to them.

The generations loop can not be parallelized, so it remained as was. We created a parallel region at the beginning of the simulation in order to avoid creating and destroying threads at each generation iteration.

With regards to the board update, the general idea was to decompose it into blocks of rows, which will then be executed at each iteration in parallel by the threads, using `omp task`'s, both for the red and black sub-generations.

The major problem with this method is the data dependency of the rows at the borders of each block, since an animal can move out of the block and there might also be a conflict there. To solve this problem, we setup the dependencies in such a way that two consecutive blocks can never run at the same time.

II Decomposition

To decompose and parallelize the simulation, we divide the board into n blocks of rows (the value of n will be explained later). Therefore, each block has $\text{int}(M/n)$ rows.

In order to process the blocks in parallel, we created a `omp task` for each one, which will be executed in parallel by the CPU. As we explained before, each block depends on the block above and below. To prevent consecutive blocks from executing at the same time, we added common dependency to each pair of consecutive blocks, in the form of an element of an array of size $n + 1$. For example, block `i` depends on `array[i]` and `array[i+1]`.

Because OpenMP adds dependencies to all previously created tasks that refer a given variable, creating the tasks sequentially for each block would not work because it would create a chain of dependencies and thus, a serial program (if block 2 depends on the execution of block 1 and block 3 on block 2, it would make it so that block 3 couldn't run before block 1 finishes).

In order to account for that, we created the tasks in an even-odd way. First, we create the tasks for all even numbered blocks, which will have no dependencies among themselves and can run in

parallel. After that, we create the tasks for the odd numbered blocks, which will depend on the even ones, so they won't execute until the two contiguous even blocks are finished. There is one final task created to account for the case where n is not a divisor of the number of rows, and so we need to update a few extra rows.

III Synchronization concerns

The main synchronization concern is that every step of the simulation completes before the previous one, namely: `Initialization > Red sub-generation > Store copy of board > Black sub-generation > Reset generation (kill foxes and increase ages) > Next generation`.

We know that using the `omp for` routine involves synchronization at the end of the loop, so synchronization is automatic for initialization (and the other part of the code where we use `omp for` routine).

This is not the case when using tasks, and so we need to add a `taskwait` clause at the end of the declaration of the task to wait for all of them to finish. Note that we did not place a `taskwait` between the two loops that create the tasks for the even and odd blocks of lines. This is because the tasks concerning the odd blocks can be executed before all the odd block tasks are finished (we can therefore use vacant threads), given that the the dependencies of each block are correctly designed.

IV Load Balancing

Regarding load balancing, the main concern was in the number of tasks to create, previously mentioned as n. Choosing fewer tasks (larger blocks), makes load balancing harder because one of the tasks might take much more time than the others, that would have to wait for it. Choosing more tasks (smaller blocks) increases the overhead in the task execution and memory accesses.

We decided to split the board into a number of blocks proportional to the number of available threads to have, at least, all threads busy. We found that splitting it by 4 times the amount of available threads gave the best results ($n = 4 \times$ `num_threads`), presenting a good trade-off between the two points mentioned above.

For the `reset_generation()` routine, since the complexity of the tasks performed in this routine depends on the type of the element found on each position (including performing no action at all in the case of finding a rock), it is naturally expected that load imbalances will occur. To account for this, the `omp for schedule(guided)` directive was used, effectively implementing a guided scheduling type for the for-loop iterations.

All other routines were considered well-balanced since they perform the same tasks on the data.

V Performance Results

We ran some instances on a lab machine (4 threads) and a laptop (8 threads), and the results are presented on Tables 1 and 2, respectively.

Table 1: Speedups for several instances on a lab machine (4 threads)

300 6000 900 8000 2e5 7 100000 12 20 9999					**4000 900 2000 1e5 1e6 10 4e5 30 30 12345**				
Serial		OMP			Serial		OMP		
T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup	T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup
26.78	26.82	11.93	11.93	2.25	110.41	110.56	49.18	48.98	2.26
26.82		11.92			110.56		48.62		
26.86		11.94			110.71		49.14		
20000 1000 800 1e5 80000 10 1000 30 8 500					**100000 200 500 500 1000 3 600 6 10 1234**				
Serial		OMP			Serial		OMP		
T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup	T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup
347.3	349.02	129.67	129.0	2.70	103.14	102.55	31.27	30.47	3.37
349.99		128.61			104.05		30.25		
349.78		128.71			100.47		29.89		

Table 2: Speedups for several instances on personal machine (8 threads)

300 6000 900 8000 2e5 7 100000 12 20 9999					**4000 900 2000 1e5 1e6 10 4e5 30 30 12345**				
Serial		OMP			Serial		OMP		
T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup	T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup
31.59	31.46	12.33	12.26	2.56	136.57	136.0	52.11	52.01	2.61
31.39		12.13			135.32		52.09		
31.41		12.32			136.1		51.83		
20000 1000 800 1e5 80000 10 1000 30 8 500					**100000 200 500 500 1000 3 600 6 10 1234**				
Serial		OMP			Serial		OMP		
T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup	T (s)	$\bar{T}$ (s)	T (s)	$\bar{T}$ (s)	Speedup
392.67	398.78	129.99	131.20	3.04	96.54	96.65	38.13	39.05	2.47
402.37		131.31			96.77		39.71		
401.3		132.3			96.65		39.32		