TÉCNICO LISBOA

Parallel and Distributed Computing

Academic Year 2021/2022

OpenMP Implementation

Group 23

Filipe Corral 81346 João Gonçalves 85211 Jiaxin Jiang 101925

Before starting to implement the parallelization we began by improving our serial version. As can be observed in table 1, the times to run the code were much faster.

Approach used for parallelization:

For the given problem it is impossible to parallelize the generations because they depend on the results obtained by the previous one. For the same reason, it was noticed that sub generation black could only be computed after the red one was finished and so this process could not be parallelized as well. Therefore we focused on parallelizing the way data was processed inside each of the sub generations. When accessing data from the arrays it was possible to divide the for cycles by the threads being used. This happens when the map is being copied between sub generations, and when the animals next possible movements are being computed and conflicts being resolved.

Decomposition used:

As explained before, a parallelization approach based on data parallelization within each sub generation was the best way forward, maximizing the amount of work being done in parallel. Considering the map is a rectangle and each cell is independent (to a degree - this will be further explored when discussing synchronization concerns) there's multiple ways of decomposing it for processing. Since the map size is usually much larger than the number of threads available and both map dimensions are similar, the division was chosen by rows. This way, the total number of rows is divided as equally as possible among the threads.

Synchronization concerns:

For he functions used to compute the copy of the map (*copy_world*), the animals next movements (*pick_cell*) and when the animals don't move (*no_movement*) there's no need to take into account data races, since the information being collected is from a map that does not change during these computations and it is impossible for threads to try to access the same memory.

For the functions responsible for computing the fox and rabbit's movements and dealing with their conflicts (*fox_movement*, *rabbit_movement*), it is possible that different animals choose to move to the same square at the same time and when dealing with these conflicts different threads might try to access the same memory simultaneously. In this case it is important to protect the data to avoid compromising the results, this is why atomic operations were used when writing to a nearby cell. It was chosen not to protect reads since all writes were protected, and read after read would never result in a data race.

Load balancing:

Since we are dividing the arrays equally for all the threads, the loads of each thread should be approximately the same. However, in a scenario where the animals are not evenly distributed in the map, some threads might have a higher work load, due to the fact that they will have more computations to make than the rest. This might make the code slower because, when badly distributed, the data will make threads idle while waiting for others to finish.

What are the performance results:

The results obtained in the lab computers are presented in table 1.

	Serial	Serial updated	OMP (1 thread)	OMP (2 threads)	OMP (4 threads)	Speed-up (4 threads)
r300	41.7	22.4	24.6	16.1	14.4	1.54
r4000	138.2	100.9	110.0	70.8	64.5	1.56
r20000	375.7	285.8	309.0	184.5	154.9	1.84
r100000	125.1	83.7	97.8	50.7	26.1	3.2

Table 1 – Performance results in seconds

We could confirm that the values decreased when parallel implementation was utilized. With the help of the *VTune* software we were able to verify that our program had parallelization rates higher that 96% in every case presented, which is optimal. The parallelization with lower number of threads is slower, as expected, and in the case of only one thread it is even slower than the serial version, since indeed it is the equivalent of a serial version with the downside of having to initiate the parallelization process.

We achieved speed-ups, for 4 threads, in a range of 1.5 to 3. These are a good range of speed-ups, however the lower values can be explained by the fact that our serial implementation is very optimized and our code is very dependant of memory operations, which we found out to be hard to optimize in the parallelization process.

Testing in a computer with 4 cores and multi-threads, achieving 8 threads in total, we obtained the values in table 2. We can verify, when comparing with 4 threads, that the speedups with 8 threads are practically the same in every case.

	Serial	**Serial updated**	**OMP (4 threads)**	**OMP (8 threads)**	**Speed-up (highest)**
r300	30.1	21.0	13.2	12.8	1.64
r4000	125.4	92.4	57.0	58.7	1.62
r20000	338.5	262.3	138.2	138.0	1.90
r100000	107.7	71.2	29.7	21.3	3.34

Table 2 – Performance results in seconds