Parallel and Distributed Computing

2021-22

Foxes and Rabbits – Ecosystem Simulator

OpenMP

Group: 10

Nelson Trindade – 93743

André Ribeiro – 104083

Georgios Stefanakis – 104596

Parallelization approach

In contrast with the serial implementation, we parallelized the computations inside a single sub-generation by distributing the rows of the world grid to the available threads. Using the OpenMP specified pragmas, each thread managed a total of $\frac{M}{P}$ row computations, where M is the number of rows in the world grid and P is the number of threads. That way, the world grid was divided into chunks of $\frac{M}{P} \times N$ sequential cells, where N is the number of columns, each of which was dispatched to a thread. This process is then repeated for the black sub-generation after the red one is completed. In this section, we faced a race condition which was handled using the *atomic capture* clause, as we describe later. After that, the program enters another parallel section in which the world grid is traversed to identify and resolve conflicts that might have appeared on cells with more than one animal. This section is also responsible for removing animals that have reached their starvation age. We use the same logic as before for distributing the required computations to the available threads. In addition, we used the collapse clause to convert the nested for loop into one loop with a larger iteration space. This allows us for better load management, something that we analyze further later in the report.

Decomposition

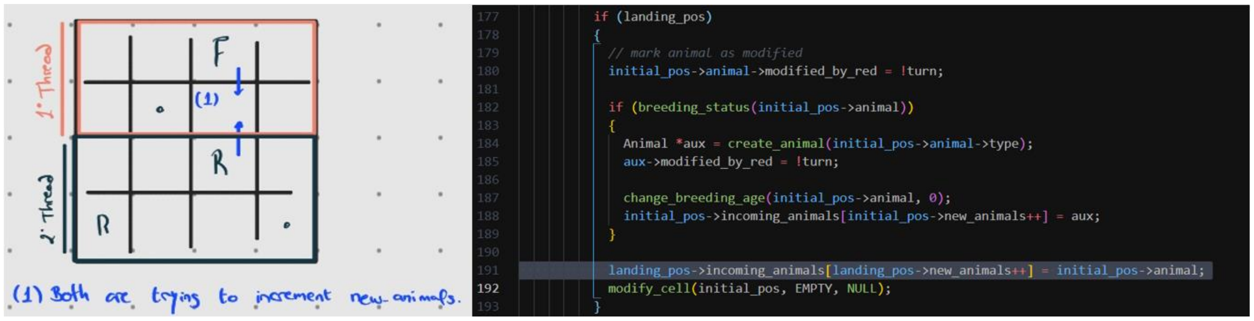
The computations that take place in a single generation of the program follow the red and black sub-generation approach, as was requested in the project announcement. Each sub-generation only performs the computations of the cells that correspond to it. A calculation of a cell consists of updating the starvation and breeding age of the animal inside it, finding the position the animal should be in the next generation, and purging the animals that have reached their starvation age. The animal is then stored in an auxiliary array that belongs to the landing cell. After both sub-generations are completed, the function *resolve_conflicts* is executed for every cell of the grid that two or more animals have landed on, to decide which of them survives the next generation.

In the serial implementation, the calculations were performed in a cell-by-cell fashion, with no concurrency between the cells of a single sub-generation. We used OpenMP to parallelize the computations in the world grid, so that the active threads execute the computations of chunks of rows in parallel, as described above. Furthermore, the *resolve_conflicts* stage was parallelized in the same way, by dividing the grid into groups of rows. The merging of the animals in different row groups is done in parallel.

Synchronization concerns

To merge the conflicts generated in the simulation, we have created an array of "incoming animals" that show the animals that fall into that cell and are then compared to the previous animal presented in that cell. This allows the conflict resolution to happen after all the calculations of the move/starvation and age are concluded and are independent of these.

Although this solution resulted in a good code structure, it creates a critical section between different threads. Whenever a cell from "above" and "below" another cell wants to add an animal to the cell in-between them (*Figure 1*), it can create a section where two threads are trying to add their animal to a shared array. With this shared array, we needed to synchronize the addition of animals to it. To do so, we first try to add a critical section, so that only one thread could execute this part of the code. By doing so, we met an increase in the time of the overall program, and it made sense. Since it was in a *for* loop, we were trying to lock a piece of code a significant number of times.

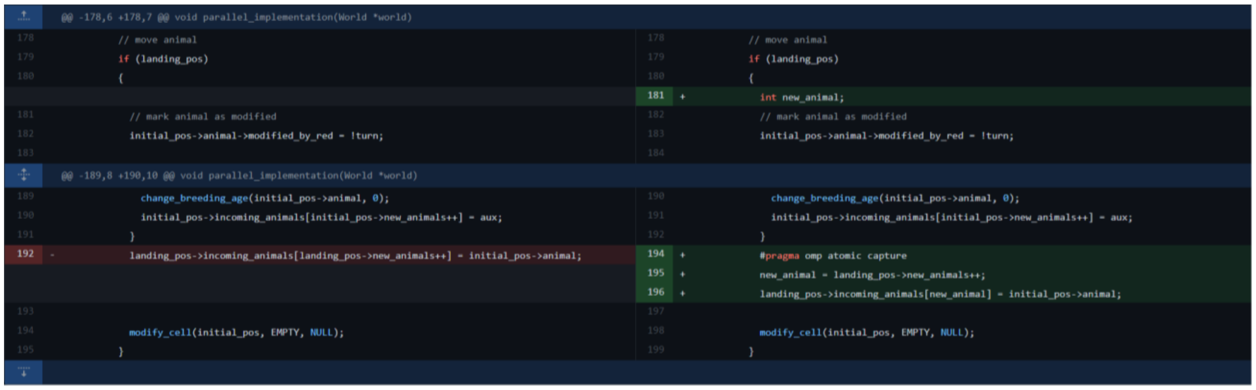


```c
if (landing_pos)
{
  // mark animal as modified
  initial_pos->animal->modified_by_red = !turn;

  if (breeding_status(initial_pos->animal))
  {
    Animal *aux = create_animal(initial_pos->animal->type);
    aux->modified_by_red = !turn;

    change_breeding_age(initial_pos->animal, 0);
    initial_pos->incoming_animals[initial_pos->new_animals++] = aux;
  }

  landing_pos->incoming_animals[landing_pos->new_animals++] = initial_pos->animal;
  modify_cell(initial_pos, EMPTY, NULL);
}
```

Figure 1 Critical Section

We research how to overcome this situation and come across the *omp atomic* command. This command "ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location"[1], so we tried it. The default setting was the *update*, but we only needed to capture the variable "new animal" for our code. The capture clause ensures that a copy of a variable is kept after the atomic operation[2].

This variable called *new_animal* keeps the index of the next unused slot in the *incoming_animal* vector so that we could add the new animal without conflicting with the other threads. If we were to use the default update option, another thread could have updated the variable before we could add the animal in the correct position. The difference is represented below:



```diff
@@ -178,6 +178,7 @@ void parallel_implementation(World *world)
       // move animal
       if (landing_pos)
       {
+        int new_animal;
         // mark animal as modified
         initial_pos->animal->modified_by_red = !turn;

@@ -189,8 +190,10 @@ void parallel_implementation(World *world)
           change_breeding_age(initial_pos->animal, 0);
           initial_pos->incoming_animals[initial_pos->new_animals++] = aux;
         }
-        landing_pos->incoming_animals[landing_pos->new_animals++] = initial_pos->animal;
+        #pragma omp atomic capture
+        new_animal = landing_pos->new_animals++;
+        landing_pos->incoming_animals[new_animal] = initial_pos->animal;

         modify_cell(initial_pos, EMPTY, NULL);
       }
```

Figure 2 Changes in a critical section

[1] OMP Atomic, by IBM
[2] OMP Atomic Capture

Load balancing

For load balancing, we use the default OpenMP static scheduling, and we managed the computation load of each thread indirectly by dividing the rows of the world grid to the available threads evenly. For the second nested *for* loop that resolves the conflicts between animals, we used dynamic scheduling for the load balancing. We also used the *collapse* clause which is something that could help the optimization of the loop. We also use the *nowait* clause since the conflict computations that take place inside a parallel section are independent of each other, and that way the performance could increase by instructing the threads not to wait for the rest iterations.

Results

After all the modifications referred to above to make our program execute in parallel, we start running tests. The tests were made with the 4 large instances provided in Fenix. The Serial Task and the Parallel Tasks with 2, 4 and 8 threads were executed in Lab 1 – PC 1 from RNL (Intel(R) Core (TM) i5-4460 CPU @ 3.20GHz, with 4 CPUs and 4 Cores). The results are in the table below:

	SERIAL V.	PARALLEL TASKS		
TEST (SEC)	**1**	**2**	**4**	**8**
R100000	165,5s	98,0s	57,5s	71,1s
R20000	779,4s	495,9s	367,6s	407,9s
R4000	214,5s	147,1s	127,3s	134,6s
R300	55,8s	37,7s	31,0s	31,5s

Table 1 Results from the larger test set (in seconds)

	PARALLEL TASKS		
SPEEDUP	**2**	**4**	**8**
R100000	1,6882	2,8808	2,3277
R20000	1,5716	2,1200	1,9108
R4000	1,4585	1,6854	1,5936
R300	1,4821	1,7990	1,7714

Table 2 Speedup Obtained

The Speedup was calculated by using $S = \frac{T_{Serial}}{T_{Parallel}}$, were T_{Serial} is the time in seconds when executing the serial project, and $T_{Parallel}$ is the time in seconds when using N Parallel Tasks, as represented in *Table 2*. It's reasonable to use this parallelization approach since when the inputs scale the speedup increases significantly. These tests were executed on a 4 Core computer, which means that the maximum performance was observed, as expected, with 4 parallel tasks. In addition, we provide below the same tests, executed on the AMD Ryzen 5900x processor, which consists of 12 physical cores @ 3.70GHz.

	SERIAL V.	PARALLEL TASKS				
TEST	**1**	**2**	**4**	**8**	**12**	**16**
R100000	65,5s	41,0s	34,8s	33,3s	33,8s	57,0s
R20000	477,7s	277,1s	184,1s	115,7s	94,7s	109,9s
R4000	43,8s	29,1s	24,2s	23,7s	22,0s	22,6s
R300	153,8s	102,0s	84,7s	66,4s	63,7s	71,9s

Table 3 Results from the larger test set (in seconds)

	PARALLEL TASKS				
SPEEDUP	**2**	**4**	**8**	**12**	**16**
R100000	1,5975	1,8822	1,9669	1,9378	1,1491
R20000	1,7239	2,5948	4,1288	5,0443	4,3466
R4000	1,5052	1,8099	1,8481	1,9909	1,9380
R300	1,5078	1,8158	2,3162	2,4144	2,1390

Table 4 Speedups Obtained

We could have gotten better results, closer to the cores of the CPUs in the lab machine of RNL, however, they were overall satisfactory results. We expect to do a better version of the upcoming MPI implementation.