INSTITUTO SUPERIOR TÉCNICO

Parallel and Distributed Computing

Foxes and Rabbits - OpenMP

Short Report

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

Group 2

April 1, 2022

Serial Version Context

Our serial approach to the ecosystem simulation consists of splitting the computation of a generation into two parts. First the red sub-generation is processed and then the black sub-generation is processed, solving conflicts as they appear. It is to note that our world is organized as a matrix of `structs`, with each `struct` holding the animal in that position and its breeding and starving age. One should note as well that there are 2 world matrices, one for the current world (start of generation) and one for the next world, where we write the changes caused by the animal movements during a generation. At the the end of each sub-generation, next world is copied onto current world. In order to not move an animal twice within a single generation, we make use of a third matrix (made of integers), used with the purpose of marking the cells of already moved animals.

Parallelization - First Approach

The only way to parallelize this type of simulation is to exploit available parallelism inside the computation of a generation, seeing as it is impossible to parallelize the computation of different generations, as every new generation directly depends on the state of the previous one. That being said, we could simply create a parallel region by using `#pragma omp parallel` each time a generation is computed. Then, inside this parallel region we could utilize `#pragma omp for` to distribute loop iterations among different threads. However, this approach has severe problems. First, and most importantly, it does not work, due to race conditions which are created by the fact that threads can attempt to write to the same position of the next world. This leads to the second problem which is that, in order to solve this, we would have to create a critical region whenever a thread tries to move an animal to another cell. This means that there would be a huge critical region which would effectively serialize the program. We gave some thought to this approach, but decided not to implement it, as we believe the performance would be very poor.

Parallelization - Second Approach

A better approach at parallelization would be to, for each sub-generation, divide the matrix into blocks, with each block containing a certain number of rows. By splitting the world this way, we can identify that computation inside each block is independent from the computation of the other blocks, meaning they can be done in parallel. Nonetheless, we must still take into account that race conditions might still exist near the margins of each block. For example, the last row of a block might need to read and write to the first row of the next block (and also the other way around), which may lead to wrong results if proper synchronization isn't applied. In order to avoid these race conditions, we came up with two ideas. The first idea was to process all rows of each block concurrently, with the exception of the first and last rows of the blocks, leaving those for later processing. The second one, which we believe is more efficient, is to avoid processing consecutive blocks of rows at the same time. Doing so eliminates the need for critical sections which would hurt performance. This is the approach we pursued and will elaborate further in the report.

In order to implement this, we resorted to assigning tasks to the threads, but carefully specifying the dependencies between them, in order to avoid concurrent execution of two coalescent blocks. The `#pragma omp task` OpenMP construct is used for the creation of each task and the dependencies between them are established by attaching the `depend` clause to the construct. For each sub-generation, we first create a set of non-coalescent tasks by using a `for` loop that creates a number of tasks equal to half the number of defined tasks. These tasks have an `out` type dependency and can run concurrently. Once all of these are created, we enter another `for` loop and create the remaining tasks, which have `in` type dependencies. These blocks are also completely independent. Figure 1 helps visualize the approach. According to what was explained, green tasks would be created first, and, afterwards, the orange tasks would be generated with dependencies to the green tasks.

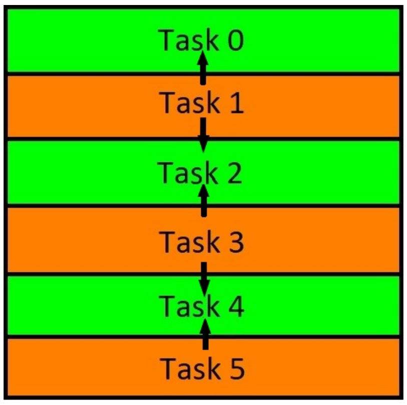

Figure 1 – Example task decomposition of a map

The number of created tasks is also very important for the performance of the program. A small number of large tasks could lead to load balancing problems, which would negatively impact performance, while a big number of small tasks can lead to large overheads due to synchronization. This being said, we conclude that there is a number of tasks, that is neither too big or too small, that optimizes the performance of our program. In order to maintain threads busy, the number of generated tasks should be at the very least double the number of threads (i.e., if we have 4 threads, we should generate at the very least 8 tasks). This means the optimal number of tasks depends on the number of threads. Empirically, we discovered that having the number of tasks equal to the number of threads multiplied by 8 leads to good performance.

Decomposition

In order to exploit parallelization, the `#pragma omp parallel` construct is called, which launches and initializes a number of threads specified by the shell variable `OMP_NUM_THREADS`. Then, as mentioned, the approach consists in creating and assigning tasks to threads which are responsible for computing a fraction of the ecosystem. Each task processes rows of the world matrix, which is represented now through a `struct` vector as, since vector data is all contiguous in memory, it makes better use of the cache hierarchy. The number of rows each task processes depends on the total number of rows and the total number of tasks. Following the rules of the ecosystem, the first round of tasks generated tasks process the red cells of its assigned rows. Then, when all tasks (and thereby all red cells) have been completed, `next_world`, which contains the changes made by the red sub-generation, is copied to `curr_world`, by using the `memcpy` function, in order to begin processing the black cells. This function is used as it is heavily optimized and therefore it efficiently copies data by working with pointers to memory regions, instead of copying element by element through nested for loops.

A new round of tasks is generated, where, once again, rows of the matrix are processed, however, now the black cells are analysed. Finally, the world is copied through parallelization by distributing iterations among the threads using the `#pragma omp for` construct and, in this same loop, we remove foxes that have starved. Note that our implementation as limitations when it comes to number of tasks created. As mentioned, its number proportionally depends on the number of running threads. Thus, and since tasks consist of a number of rows that have to be computed, each task must have at least two rows to avoid race conditions. In fact, if each task in figure 1 corresponded to a single row, then different green tasks could be writing to the same cells simultaneously, which is problematic. We assume that the number of threads is carefully chosen in order not to cause the mentioned scenario.

Synchronization

Introducing and exploiting parallelism often induces wrong computations that wouldn't occur in sequential execution, as now instructions may be executed out of order (ignoring dependencies that might exist) and race conditions become a possible scenario (leading to wrong calculations). Although distributing workloads across different threads allows reducing execution times, the need to synchronize the program in situations where hazards may surge is mandatory, which comes with the cost of hurting the program's potential performance (specially when critical regions are applied). Therefore, a program is desired to have as least synchronization as possible while maintaining its correct functioning, and restructuring the program's code often allows to decrease its amount. For instance, we wish that no consecutive tasks are processed simultaneously (to avoid having to apply critical sections at the boundaries of each block, where race condition exists between cells), and thus, we define a set of dependencies for each created task. By making a certain task dependent on its adjacent tasks, we ensure that it won't ever be necessary to lock cells of the matrix, which speeds up the execution. Nonetheless, after processing a sub-generation, a copy of the world must be made. This means that all threads should wait for the completion of all tasks before beginning the copying process. For the world copy after the red sub-generation, synchronization is implicit by the `#pragma omp single` construct, while, for the world copy after the black sub-generation, synchronization is applied through an implicit barrier imposed by the `#pragma omp for` construct.

Load Balancing

When distributing workloads across different threads, it becomes necessary to do so in a balanced way, in order to maximize the computational resources and to avoid leaving the processor's cores (threads) idle. For example, giving a strong workload to a single thread and weaker ones to the remaining ones (and assuming they all synchronize after finishing all the work) leaves these threads bounded by the one that has the large amount of computation to do. Work unbalance may occur when different loop iterations or tasks, due to divergent execution, for example, leads to different amounts of instructions to be executed in those iterations. To prevent this from happening we create enough tasks for all the available threads so that whenever a thread (to which a certain task was assigned) finishes, it can be assigned to another task and therefore avoiding being idle.

Results

Table 1 contains the execution time of different tests ran on the `lab4p6` machine to benchmark the performance of our OpenMP implementation of the project. The table shows the execution times and speedups obtained for different numbers of threads and for the following testing maps.

1. Test 1 = 300 6000 900 8000 200000 7 100000 12 20 9999
2. Test 2 = 4000 900 2000 100000 1000000 10 400000 30 30 12345
3. Test 3 = 20000 1000 800 100000 80000 10 1000 30 8 500
4. Test 4 = 100000 200 500 500 1000 3 600 6 10 1234

Table 1 – Execution times and speedups obtained for different tests and different numbers of threads

	Test 1 [s]	Test 2 [s]	Test 3 [s]	Test 4 [s]
Serial	18.58	78.37	275.58	73.18
2	13.02 (1.43x)	54.08 (1.45x)	181.89 (1.52x)	49.62 (1.47x)
4	9.55 (1.95x)	40.45 (1.94x)	120.17 (2.29x)	30.91 (2.37x)
8	9.93 (1.87x)	43.53 (1.80x)	129.33 (2.13x)	38.80 (1.89x)

As we can see, the obtained speedups are well below the ideal ones. Ideally, the speedup would be equal to the number of threads. The fact that the obtained speedups aren't ideal, can be explained by the overhead of creating multiple tasks and of context switching of threads. Speedup is also limited by the synchronization that is imposed by the inter-task dependencies and by the world copying. Something else that is interesting to note is that the speedup obtained for 8 threads is lower than the speedup obtained for 4 threads. Considering that the processor the program is running on is a 4-core processor with hyper-threading support, when the number of threads is 8, we have two threads sharing a core. That being said, those two threads would now be competing for resources like bandwidth and cache, which can outweigh the benefits of the extra number of threads.