Project Assignment II: Ecosystem Simulation

Emanuel Rodrigues de Sousa Tomé[1]

[1] *MSc Data Science, Department of Computer Science, Faculty of Sciences, University of Porto*

13th January 2021

1 Introduction

The present work's goal is to implement the Ecosystem Simulation problem using parallel programming for shared-memory environments. A sequential version of the application was developed (ecosystem_seq.c) and then a parallel version was implemented using OpenMP (ecosystem_parallel.c). To compile the developed parallel program, the following command should be done in the terminal:

```
$ gcc -o ecosystem_par -fopenmp ecosystem_parallel.c
```

To run the parallel version, the following syntax should be followed:

```
$ ./ecosystem_par <input_file> <number_of_threads> <chunk_size (optional)>
```

For instance, for the input file "input100x100" with 2 threads and a chunk size of 50, the following command should be executed:

```
$ ./ecosystem_par input100x100 2 50
```

Note that the chunk size is optional. If it is not given, the algorithm will always consider that the chunk size is equal to the ecosystem world size divided by the number of threads.

2 Implemented Algorithm

2.1 General description

The developed algorithm to solve the Ecosystem Simulation problem is composed of:

- The macros **MIN**, **MAX** (to determine the minimum and maximum value between two values, respectively) and **matrix** (to easily handle matrices).
- The structure types **ECOSYSTEM** (where the characteristics of the current ecosystem are stored) Moreover, **COORDS** (where the free positions to move of the position being analysed are stored).

```
// Definition of the structure type ECOSYSTEM
typedef struct {
    int gen_proc_rabbits;   // number of generations until a rabbit can procreate
    int gen_proc_foxes;     // number of generations until a fox can procreate
    int gen_food_foxes;     // number of generations for a fox to die of starvation
    int n_gen;              // number of generations for the simulation
    int r;                  // number of rows of the matrix
 representing the ecosystem
    int c;                  // number of columns of the matrix representing the ecosystem
    int n;                  // number of objects in the initial ecosystem
    int *world;             // ecosystem world
} ECOSYSTEM;

// Definition of the structure type COORDS
typedef struct {
    int row[4];     // row index of the free positions to move
    int col[4];     // column index of the free positions to move
    int n_free;     // free positions to move available
} COORDS;
```

- The functions:

- **read_ecosystem** - to read the input files. Note that the objects are encoded as follows: 0 - empty cell, 1 - cell with a rabbit, 2 - cell with a fox and 3 - cell with a rock;
- **print_ecosystem** - to print the ecosystem world to the stdout console (this is a debugging function, and it is deactivated in the final compilation of the developed program);
- **copy_ecosystem** - to duplicate the structure type ECOSYSTEM;
- **copy_matrix** - to make a duplicate of a matrix;
- **out_ecosystem** - to print to a file the final ecosystem after the finish of all computations;
- **out_maps** - to output the current ecosystem world map, procreation map and food map to a file (this is a debugging function, and it is deactivated in the final compilation of the developed program);
- **pos_to_move** - to determine the available positions to move. If looking for empty cells, the variable int_to_look is set equal to 0, if looking for cells with rabbits, int_to_look should be set equal to 1. This function is called by functions move_rabbits and move_foxes. The latter is used twice: first to determine if there are rabbits in the adjacent cells and second to determine the available empty cells.
- **move_rabbits** - This function, together with the function core_rabbits, is where the rabbits' movements take place. The ecosystem world matrix is looped over its rows and columns to determine the new positions of the rabbits. This is also where, together with the function move_foxes, the parallelization occurs in the developed algorithm as detailed in the next section.
- **core_rabbits** - This function is called inside the function move_rabbits. In this function, all the problem rules related to movement and procreation of the rabbits are defined and take place in the ecosystem world.
- **move_foxes** - This function, together with the function core_foxes, is where the movements of the foxes occur. The ecosystem world matrix is looped over its rows and columns to determine the new positions of the foxes. This is also where, together with the function move_rabbits, the parallelization occurs in the developed algorithm as detailed in the next section.
- **core_foxes** - This function is called inside the function move_foxes. In this function, all the problem rules related to the movement, procreation and die of starvation of the foxes are defined and take place in the ecosystem world.
- **main** - The main function is where the program is initialised and all the other functions are called. Note that while the number of generations for the simulation (variable ecosystem.n_gen) is bigger than zero, the functions move_rabbits and move_foxes are repeatedly called.

2.2 Relevant auxiliary functions and parallelization

The most relevant auxiliary functions of the developed program are the functions **move_rabbits**, **core_rabbits**, **move_foxes** and **core_foxes**. As referred before, the functions **core_rabbits** and **move_foxes** is where all the ecosystem rules are defined for rabbits and foxes, respectively. Reading the position of the actual generation in the variable ecosystem, the rabbits and foxes are placed in their positions in the generation being computed in the variable ecosystem_2. Regarding the parallel implementation, this take place in functions **move_rabbits** and **move_foxes**. The parallelization of both functions is similar, so only a excerpt of the function **move_rabbits** is presented bellow:

```
(...)
int n_loc = 0; // local number of objects
int k_for1, k_for3;
#pragma omp parallel num_threads(NTHREADS) shared(ecosystem, ecosystem_2, gen, proc_rabbit,
    proc_rabbit_2, chunk) {
    // first for loop (solving the cores of chunks)
    #pragma omp for schedule(dynamic, 1) reduction(+: n_loc) \
    private(coords_free, row, col, final_pos)
    for (k_for1 = 0; k_for1 < ecosystem->r; k_for1 += chunk){
        for (row = k_for1 + 1; row < MIN(k_for1 + chunk - 1, ecosystem->r - 1); row++) {
            core_rabbtis(ecosystem, &ecosystem_2, coords_free, final_pos, &n_loc,
                         proc_rabbit, proc_rabbit_2, row, col, gen);
        }}
    // second for loop (solving the first and last line)
    #pragma omp for schedule(dynamic, 2) reduction(+: n_loc) \
    private(coords_free, row, col, final_pos) nowait
    for (row = 0; row < ecosystem->r; row += ecosystem->r-1) {
        core_rabbtis(ecosystem, &ecosystem_2, coords_free, final_pos, &n_loc,
                     proc_rabbit, proc_rabbit_2, row, col, gen);
    }
    // third for loop (solving the boundaries)
    #pragma omp for schedule(dynamic, 1) reduction(+: n_loc) \
```

```
    private(coords_free, row, col, final_pos)
    for (k_for3 = chunk-1; k_for3 < (ecosystem->r-1); k_for3 += chunk){
        for (row = k_for3; row < k_for3+2; row++) {
            core_rabbtis(ecosystem, &ecosystem_2, coords_free, final_pos, &n_loc,
                         proc_rabbit, proc_rabbit_2, row, col, gen);
        }}}
// Update number of objects
ecosystem_2.n += n_loc;
(...)
```

The main idea of the parallel implementation is to divide the ecosystem world's lines by the different threads. The block of lines to be solved by each thread is defined in the variable **chunk**, which can be defined by the user when they start the program (please, see details of the input parameters in the section introduction). However, care should be taken in the boundaries of the blocks of lines. Therefore, in the first loop, only **chunk-2** lines are computed. The boundary lines are solved after in the third for loop, which is also parallelized. Note that each thread will solve two boundary lines. The first and last line are also solved by one thread in the second for loop. Note that since those lines' computation does not interfere in the results obtained in the boundary lines, the directive **nowait** is used.

2.3 Performance evaluation

In the following tables, the sequential and parallel execution times, the obtained speedups and the efficiency are presented. As expected, in general the sequential version's processing time is lower than the parallel version's processing time using only one thread, particularly for the smaller ecosystems. Note also that for those ecosystems (of size 5, 10 and 20), the parallel execution time is bigger when the number of threads is bigger. Therefore, it can be concluded that communication time has significant importance in the total execution time for small ecosystems. For the bigger ecosystems, shorter execution times were generally obtained when the number of threads increased. However, the efficiency decreases drastically, and the application is not scalable since the efficiency for 2 threads, and an ecosystem size of 100 is 0.65 and for 4 threads and an ecosystem size of 200 is 0.38 (see 4). Finally, it should also be mentioned that the execution time may be sensitive to the chosen chunk size. The present analysis only presented the obtained execution times for a chunk size of the number of rows divided by the number of threads. However, it should be mentioned that slight variations were observed when different chunk sizes were considered (see table 5).

Table 1: Sequential execution time (ms)

Ecosystem						
5	10	20	100	200	100_u1	100_u2
0.04	1.70	44.72	10 166.48	42 881.93	8 719.24	9 988.86

Table 2: Parallel execution time (ms)

No. threads	Ecosystem						
	5	10	20	100	200	100_u1	100_u2
1	0.12	2.42	41.88	10 173.70	42 550.62	8 684.32	10 003.41
2	0.37	3.99	51.77	7 779.68	33 627.78	7 205.24	7 757.93
4	0.63	5.32	64.76	6 536.31	27 875.39	6 234.92	6 451.72
8	1.04	7.36	83.82	5 819.85	25 045.67	5 922.67	5 821.23
16	1.89	9.47	131.67	5 938.44	24 759.09	6 107.60	5 824.14

Table 3: Speedups

No. threads	Ecosystem						
	5	10	20	100	200	100_u1	100_u2
1	1	1	1	1	1	1	1
2	0.310	0.608	0.809	1.308	1.265	1.205	1.289
4	0.186	0.456	0.647	1.556	1.526	1.393	1.551
8	0.112	0.329	0.500	1.748	1.699	1.466	1.718
16	0.061	0.256	0.318	1.713	1.719	1.422	1.718

Table 4: Efficiency

No. threads	Ecosystem						
	5	10	20	100	200	100_u1	100_u2
1	1	1	1	1	1	1	1
2	0.16	0.30	0.40	0.65	0.63	0.60	0.64
4	0.05	0.11	0.16	0.39	0.38	0.35	0.39
8	0.01	0.04	0.06	0.22	0.21	0.18	0.21
16	0.00	0.02	0.02	0.11	0.11	0.09	0.11

Table 5: Execution time using different chunk sizes (ms)

No. threads	chunk	Ecosystem		
		100	100_u1	100_u2
2	50	7 780.86	7 337.00	7 779.68
	25	7 711.99	7 312.15	7 699.72
	20	8 083.80	7 287.54	7 984.68
	15	7 814.85	7 261.06	7 794.13
	10	7 778.45	7 250.06	7 766.75
4	25	6 368.01	6 331.48	6 487.69
	10	6 605.90	6 402.78	6 579.59

2.4 Main difficulties encountered

In the first stage, the main difficulties encountered in developing the work were correctly implementing all the rules. For instance, it was not clear for the author that the reproduction index should continue to increase when it reaches the number of generations for procreation and the rabbit or fox cannot move. In a second stage, the author started to parallel algorithm using the locking functions of the OpenMP. However, for small chunk sizes, the program started to be sequential, and because of that, the approach presented in the previous sections was preferred. It should also be mentioned that although the parallel implementation of the developed algorithm has shorter execution times than the sequential version for bigger ecosystem worlds, the developed algorithm is not scalable.

3 Conclusions

The Ecosystem Simulation problem was, as intended, solved using parallel programming for shared-memory environments using OpenMP. In the present report, the main points of the developed algorithm were described. A performance evaluation was also presented for different sizes of the ecosystem world and different chunk sizes. It was demonstrated that the sequential version of the implemented algorithm is faster for small ecosystem worlds, which was expected. For bigger ecosystem worlds, it was demonstrated that the parallel version has shorter execution times than the sequential version. However, the proposed parallelization is not scalable.