Simulation of Life Evolution

EduHPC'20 Peachy Assignment

1 Introduction

You are provided with a sequential code that simulates the evolution of simple life forms (protozoa) in a humid surface or culture. Only a rectangular surface is considered in the simulation. This surface is modelled as a set of discrete and evenly spaced points or positions. A bidimensional array is used to represent the amount of food on each point in a given time step. On each simulation iteration, more food is randomly spread on the surface. The number of points receiving new food and the quantity is randomly chosen according to some program arguments. We can define scenarios with a special square zone or spot, where food is spread with more probability and in more quantity.

Another one-dimensional array is used to store the information of a set of protozoa/bacteria/cells that move on the surface in order to collect food. The information stored for a cell includes: Its age (in simulation steps), its position, a unit vector indicating the movement direction, the amount of accumulated food and its genetic code, three values that describe the probability to keep moving straight-forward, turning left, or turning right. On each simulation iteration a random number is chosen and compared to the genes values to keep or change the movement direction of the cell.

When a cell arrives at a position, it gathers all the food on it. If several cells are in the same position, they share the food splitting it evenly. The cells spend the precious food they have gathered to keep moving, one unit of food per iteration. If they don't have enough food, they stop, starve and finally die. The simulation stops if all the cells die. But if a cell manages to reach maturity (a given number of steps alive), and it has got enough food, it splits in two new young cells with age one, half the accumulated food of the original cell, and random starting directions. In the splitting process, the original genes are slightly mutated, randomly changing the movement probabilities of the new cells. After enough simulation iterations it is possible to observe the effect of natural selection. Most of the surviving cells are adapted to the environment, and most of them show specific patterns of gene values.

Figure 1 shows the output of the program for a surface of 30x30 points, after 3 000 iterations. There is an accumulation of cells with high values of turning genes around a special spot where a lot of food is provided. Some other errant cells with high values in the straight-forward gene traverse the culture avoiding to exhaust a specific region. The following symbols are used to represent the state of each control point:

`( n )`: There are n cells in the position.

·: There are 5 or more units of food in the position.

`*`: There are 10 or more units of food in the position.

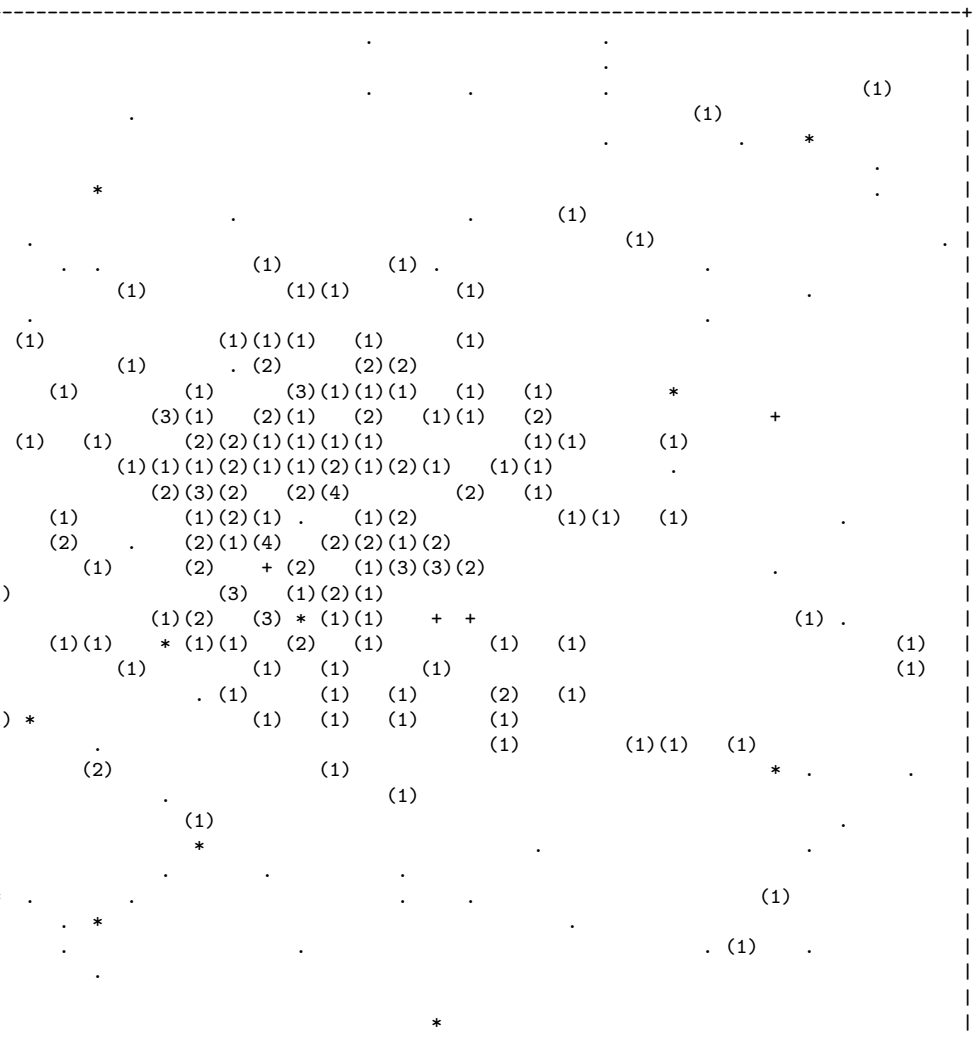
`+`: There are 20 or more units of food in the position.

2 Sequential code description

Program arguments The program arguments describe the simulation scenario:

```
Arguments, Rows: 40, Columns: 30, max_iter: 2999
Arguments, Max.food: 120.000000, Food density: 0.012000, Food level: 12.000000
Arguments, Init Random Sequence: 5324,6546,3243
Arguments, Food_spot, pos(15,6), size(10,10), Density: 0.110000, Level: 25.000000
Initial cells: 80

Iteration: 2999
```



```
Num_cells_alive: 0187
History( Cells: 8476, Dead: 8289, Max.alive: 0289, Max.new: 0048, Max.dead: 0012,
         Max.age: 0156, Max.food: 96.109299 )
```

Figure 1: Example of the program output in debug mode for the indicated argument values.

1. rows, columns: Number of control points in the culture surface (size of the array to store the food value in each point).

2. `max_iter:` Maximum number of simulation iterations to execute.

3. `max_food:` Maximum level of food in a position. The simulation stops if a point accumulates more food.

4. `food_density:` Value between 0 (none) and 1 (all), indicating the amount of random points where food is spread on a simulation iteration.

5. `food_level:` Maximum quantity of random food that is spread on a point.

6. `short_rnd1, 2 and 3:` Three unsigned short numbers that are used as seeds to initialize random generators for the food spread and cells actions. Thus, the result of the simulation for the same collection of arguments is reproducible.

7. `num_cells:` Number of initial random cells in the culture.

8. Optional: Data for a special square spot where more food is spread: Position of the top-left corner, height and width, food density, and food level.

Students are encouraged to create their own scenarios, to test different situations and problems. Several examples are provided along with the code.

Results The program shows at the end of the simulation the following results. The execution time of the simulation (without initialization times), and a list of statistical data that is used to verify the correctness of the simulation.

Debug mode If the program is compiled with the flag -DDEBUG (included in the provided makefile) a piece of code is activated to print with ASCII art a representation of the surface at the end of each iteration (as in the previous figure). It can help to detect errores, simulation deviations, or simply to show its evolution to obtain a better idea of how it works. The data of the cells at the start and end of the simulation are also shown. Thus, the resulting values of the genes can be observed.

A note about random numbers and parallelism The generation of a pseudo-random sequence of numbers with the classical C library functions is inherently sequential. Our program uses library functions that compute the next random value using a small input array that stores the state of a random sequence after each call. These functions can be used to obtain values of different sequences identified by their state arrays. Thus, several random sequences for food and for each cell actions can be used in the program independently. Nevertheless, take care to avoid changing the order in which random numbers from a given sequence are obtained, or the simulation results will change and the program will be non correct.

3 Project goal

Use the parallel programming model proposed by the teacher to parallelize this program, optimize the code, and obtain the best possible performance.

Code modifications allowed Students can modify the sequential code provided as long as they observe the following restrictions:

- Exploit parallelism using only the parallel programming model proposed.
- The argument processing section, array memory allocations, time measurement points, and output of results, should not be changed. The section of code that the students should target and modify is clearly identified in the main function. This section is found between the points where the time measurement is started and ended. Functions defined in the program that are called from the target section of the code can also be modified, substituted or eliminated.
- Any change in the algorithm or data structures must be discussed with the teachers in advance, in order to avoid modifications that significantly alter the parallelism exploitation with the proposed parallel programming model, which is the purpose of the assignment.

To measure execution times, compile the program with the maximum optimization level of the compiler (for example gcc -O3). We want to focus on program changes that do not interfere, and even facilitate, compiler optimizations.