TÉCNICO LISBOA

Parallel and Distributed Computing

Project Assignment

GAME OF LIFE 3D

Version 1.1 (16/02/2024)

2023/2024

3rd Quarter

Contents

Revisions

Version 1.0 (February 6, 2024)	Initial Version
Version 1.1 (February 16, 2024)	Fixed MPI deadline

1 Introduction

The purpose of this class project is to gain experience in parallel programming on shared and distributed memory systems, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of the game of life in 3D.

2 Problem Description

The life game is a zero-player game, that starts from an initial distribution of individuals in the game space. The game evolves by computing new generations of individuals based on simple rules.

For this assignment we consider a three-dimensional space, a cube with N cells per side. Each cell will have 26 neighbor cells (this is true for all cells, we consider that the sides wrap-around, *i.e.*, cells with indices i and $i + N$ are the same) . Each cell of this cube may be occupied (live) or empty (dead). In order to determine the status of a cell in the next generation, the following rules should be used:

- a live cell with 4 or less live neighbors dies.
- a live cell with 5 to 13 live neighbors lives on to the next generation.
- a live cell with more than 13 live neighbors dies.
- a dead cell with 7 to 10 live neighbors becomes a live cell.

Additionally, consider that we may have up to 9 species in the game, thus the cells will hold a value between 1 and 9. The species of a cell that becomes alive corresponds to the majority of the species surrounding it (to break ties, use the lower species value in the tie).

From an initial population in the cube, the objective is to compute the population through a given number of generations. Note that all the cells of the next generation are computed from the cells of the previous iterations, hence the order that the cells are computed is irrelevant.

To avoid reading (and moving around) large files, the initial population is generated randomly inside the program. To ensure that everyone has the same starting point (allowing for the validation of the results), the routine to create the simulation space is given in Appendix A. You are free to modify this this code, but please make sure you get the same initial population!

3 Implementation Details

3.1 Input Data

Your program should allow exactly 4 input line parameters, in this order:

- number of generations (positive integer)
- number of cells per side of the cube (positive integer)
- density of the initial population (float between 0 and 1)
- seed for the random number generator (integer)

3.2 Output Data

The program should send to the standard output (`stdout`) 9 lines, one per species and in increasing order of these, each with three integers representing, respectively: the species id; the maximum number of these species over all generations; generation where this maximum was achieved (lowest generation in case of draw).

Note that we will be using very large simulation spaces, be sure to use **integers with 64 bits** for your counters.

Your program should send these output lines (and **nothing else!**) to the standard output.

The project **cannot be graded** unless you follow these input and output rules!

3.3 Sample Problem

If you run your program with the following input parameters, the result should be what is presented next:

```
$ ./life3d 4 4 .4 100
1 4 3
2 1 0
3 2 0
4 1 0
5 1 0
6 14 4
7 5 0
8 2 0
9 4 0
```

For debugging purposes, you can check the population evolution for this example in Appendix B.

3.4 Measuring Execution Time

To make sure everyone uses the same measure for the execution time, the routine `omp_get_wtime()` from OpenMP will be used, which measures real time (also known as "wall-clock time"). This same routine should be used by all three versions of your project.

Hence, your programs should have a structure similar to this:

```
#include <omp.h>
<...>

int main(int argc, char *argv[])
{
    double exec_time;

    grid = gen_initial_grid(N, seed, density);
    exec_time = -omp_get_wtime();

    simulation();

    exec_time += omp_get_wtime();
    fprintf(stderr, "%.1fs\n", exec_time);
```

```
    print_result();        // to the stdout!
}
```

In this way the execution time will only account the algorithm running time, and be sent to the standard error, `stderr`. The use of these two output streams allows the validation of the results and analysis of execution time needed to be performed separately.

Because this time routine is part of OpenMP, you need the include `omp.h` and compile all your programs with the flag `-fopenmp`.

4 Submission

4.1 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C (or C++). Name the source file of this implementation `life3d.c`. As stated above, your program should expect exactly four input parameters.

This will be your base for comparisons and it is expected that it is as efficient as possible.

4.2 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `life3d-omp.c`. You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization more effective and more scalable. Be careful about synchronization and load balancing!

4.3 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `life3d-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to take into account the minimization of the impact of communication costs. You are encouraged to explore different approaches for the problem decomposition.

Note that this distributed version should permit running larger instances, namely instances that do not fit in the memory of a single machine.

Extra credits will be given to groups that present a combined MPI+OpenMP implementation.

4.4 What to Turn In

You must eventually submit the sequential and both parallel versions of your program (**please use the filenames indicated above**), and a table with the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks for both OpenMP and MPI, and additionally 16, 32 and 64 for MPI).

For both the OpenMP and MPI versions (not for serial), you must also submit a short report about the results (2-4 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why

- how was load balancing addressed
- what are the performance results, and are they what you expected

The code, makefile and report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>serial.zip`, `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

One final note, the parallel implementations should work with the OpenMP and MPI versions installed in the lab computers.

4.5 Deadlines

1st due date, serial version: **March 2nd**, until 23:59.

2nd due date (OpenMP): **March 16th**, until 23:59.

3rd due date (MPI): **March 30th**, until 23:59.

A Code to Generate the Initial Grid

```
#define N_SPECIES 9

unsigned int seed;

void init_r4uni(int input_seed)
{
    seed = input_seed + 987654321;
}

float r4_uni()
{
    int seed_in = seed;

    seed ^= (seed << 13);
    seed ^= (seed >> 17);
    seed ^= (seed << 5);

    return 0.5 + 0.2328306e-09 * (seed_in + (int) seed);
}

char ***gen_initial_grid(long long N, float density, int input_seed)
{
    int x, y, z;

    grid = (char ***) malloc(N * sizeof(char **));
    if(grid == NULL) {
        printf("Failed to allocate matrix\n");
        exit(1);
    }
    for(x = 0; x < N; x++) {
        grid[x] = (char **) malloc(N * sizeof(char *));
        if(grid[x] == NULL) {
            printf("Failed to allocate matrix\n");
            exit(1);
        }
        grid[x][0] = (char *) calloc(N * N, sizeof(char));
        if(grid[x][0] == NULL) {
            printf("Failed to allocate matrix\n");
            exit(1);
        }
        for (y = 1; y < N; y++)
            grid[x][y] = grid[x][0] + y * N;
    }

    init_r4uni(input_seed);
```

```
    for (x = 0; x < N; x++)
        for (y = 0; y < N; y++)
            for (z = 0; z < N; z++)
                if(r4_uni() < density)
                    grid[x][y][z] = (int)(r4_uni() * N_SPECIES) + 1;

    return grid;
}
```

B Debug of Simple Input

Your program **should not** print this, use this information only for debug purposes.

```
Generation 0   ------------------------------
Layer 0:
    1 6
  1 8
  6   4
    9

Layer 1:
6
1   2
3   9 7
6

Layer 2:
  3 9 6

9   6
6

Layer 3:
  6   6
6 7 7
  8 7 5
7

Generation 1   ------------------------------
Layer 0:
6   1 6

      4
    9

Layer 1:
6   1
1   2
3   9 7
6

Layer 2:
  3 9 6

9   6
6
```

```
Layer 3:
6 6   6
6 7
6 8 7 5
7

Generation 2   ------------------------------
Layer 0:
6   1 6

    7 4
    9

Layer 1:
6   1
1   2
3 6 9 7
6

Layer 2:
  3 9 6

9   6


Layer 3:
6 6 6 6
6 7
6 8 7 5
7

Generation 3   ------------------------------
Layer 0:
6   1 6

    7 4


Layer 1:
6 1 1
1   2
3 6 9 7
6

Layer 2:
  3 9 6
```

```
9   6


Layer 3:
6 6 6 6
6
6 8 7 5
7

Generation 4   ------------------------------
Layer 0:
6   1 6

    7 4


Layer 1:
6 1 1 6
1   2
3 6 9 7
6

Layer 2:
  3 9 6

9   6


Layer 3:
6 6 6 6
6
6 8 7 5
7
```