Instituto Superior Técnico

Professors:
José Monteiro
Afonso Alemão
Catarina Bento
Miguel Graça

Course:
Parallel and Distributed Computing

Parallel and Distributed Computing - Game Of Life 3D OpenMP Delivery

Group 27

Authors:

Tomás Marques, 99340
Miguel Mano, 99286
João Ivo, 90115

March 30, 2024

1 Introduction

This report, made in the context of the course Parallel and Distributed Computing's project OpenMP assignment, serves to explain our approach to the parallelization and redesign of the previous serial version. The project is based on the Game of Life with a 3D matrix composed by cells either dead or alive, carrying a value corresponding to one of the species in case of the cell being alive. Our task was to iterate over the 3D Matrix cube and simulate new generations. First we're going to talk about our approach in the serial version delivery, and some changes made to it at this moment, then we'll see the *OpenMP* instructions used and why, and finally check the results we obtained.

2 Serial Changes

In the first delivery of the project, the serial version, we tried to optimize it as much as we could, by iterating over the entire cube the minimum times we could, so in terms of loops, only threads would improve it. When we started using with *OpenMP* in the second delivery of the project, after we had parallelized the only iteration over the cube, the resulting speedup wasn't around what was expected (around the number of threads we were using, since it was only one loop), so we though it could be because of time lost when the threads were disputing over parts of memory. After analysing through our serial version, we noted that we were allocating space for the values *int** array with 2 integers, that would receive the information needed to calculate the cell's value for the next generation (one integer for the number of living neighbors and other for the majority of species present in the neighbors). Since every thread would be allocating memory for this array, in each cell iteration, it was causing a memory overload that was slowing our execution and blocking our parallelization. We fixed this by moving or removing unnecessary memory allocation that would have a some impact in the system's load balance, in our case variables inside of our parallel zone.

```
void simulation(char*** grid, char*** new_grid, long long n_cells, long long generation, long long max_generations) {
    long long species[N_SPECIES + 1] = {0};

    if (generation > max_generations) {
        return;
    }

    pair values;
    long long i, j, k;

    for (i = 0; i < n_cells; i++) {
        for (j = 0; j < n_cells; j++) {
            for (k = 0; k < n_cells; k++) {
                values = compute_neighbors(grid, n_cells, i, j, k, values);
                species[grid[i][j][k]]++;
                new_grid[i][j][k] = compute_value(values.live_neighbors, grid[i][j][k], values.majority);
            }
        }
    }
    store_output_data(generation, species);
    simulation(new_grid, grid, n_cells, generation + 1, max_generations);
}
```

Figure 1: New simulation function.

As the function *compute_ neighbors()* is now going to return a *pair structure*, we needed to create an instance of this *struct* before the for loop cycle is started to then reuse the same instance of values, in order for the threads to run in a less expensive way, in terms of memory allocation and time consumption, we opted to allocate the *struct* before the generation iteration, and then just pass it as an argument to *compute_ neighbors()*, that returns a new changed version of it and replace the current values inside simulation.

```
pair compute_neighbors(char*** grid, long long n_cells, long long x, long long y, long long z, pair values)
```

Figure 2: New compute_neighbors function.

We've tested the performance by measuring the execution times based on the serial version with the changes we discussed just now and this final version was the one where we could obtain lower execution times and betters speedups.

2.1 Updated After 2nd Delivery's Feedback

After receiving the feedback for this second delivery, our serial version wasn't as optimized as it could be, so we began to dig more in, and found about some possible lost performance cases.

2.1.1 Unnecessary iteration, to compute majority of species

With the feedback from the teacher after the delivery, we though that the loss of performance could be associated with an unnecessary iteration over the neighbors values, to check the majority of species present in the sample, since we only need to check the majority value for a dead cell that would have the correct amount of living neighbors to be be resurrected in the next generation. However this didn't fix our performance issues, despite the fact that less instructions were now being performed.

```
if (grid[x][y][z] != 0 || values.live_neighbors > 10 || values.live_neighbors < 7) {
    values.majority = 0;
    return values;
}
```

Figure 3: Verification before calculating majority.

2.1.2 Wrap-around calculation

As our problem wasn't fixed, we continued to look for a solution and found a possible loss of performance caused by the modulo (%) operation, caused by either compiler optimizations, branching, divisions that happens underneath this operation, or even pipeline stalls, as we were using it in the calculation of the neighbors cells, we've changed it to a solution that computed the neighbors coordinates in 2 different ways without the modulo:

- Cubes that the side is a power of 2 : By doing this operation $(x+i+n\_cells)$ & $(n\_cells-1)$, if n_cells is power of 2, all its bits will be set to 1, so this bitwise AND operation, will have the same result as a module operation, but in a more efficient way;
- Rest of all the other cases : in this cases instead of computing the neighbor's coordinate with module, we change it to verify if the neighbor was outside the loop, and perform the wrap-around by either adding or removing the number of cells per side of the cube.

In the beginning of the next page there is the new wrap around calculation.

```
if (CELLS_POWER_TWO) {
    n_x = (x + i + n_cells) & (n_cells - 1);
    n_y = (y + j + n_cells) & (n_cells - 1);
    n_z = (z + k + n_cells) & (n_cells - 1);
} else {
    n_x = x + i;
    n_y = y + j;
    n_z = z + k;

    if (n_x < 0)
        n_x += n_cells;
    else if (n_x >= n_cells)
        n_x -= n_cells;

    if (n_y < 0)
        n_y += n_cells;
    else if (n_y >= n_cells)
        n_y -= n_cells;

    if (n_z < 0)
        n_z += n_cells;
    else if (n_z >= n_cells)
        n_z -= n_cells;
}
```

Figure 4: New wrap-around calculation.

3 OpenMP Approach

After going through our for loop cycles, trying to parallelised them and improve our program's execution time, we had to decide which zones should be converted into parallel ones. As explained before, in our serial version we only had one iteration over the entire 3D matrix per generation, in the *simulation()* method, this was the zone where we could improve our performance the most, since the other three for loop presents in the program's execution are :

- a loop to allocate the space for the copy of the cube, we first tried to parallel this allocation loop, however the speedup for larger samples was not being as expected (as the threads were probably blocking each other from accessing memory);
- another loop to go through the neighbors of a cell, in order to compute its value for the next generation (26 iterations per cell), but as it is already inside the simulation parallel zone, we decided it was better not to try and parallel it more;
- the last one is checks and stores data about the current generation (9 iterations per generation), as the number of iterations is relative small we decided not to parallel it.

After this short explanation on what cycles to parallel, let's move into the parallelization made by us in the *simulation()* method.

```
#pragma omp parallel for collapse(2) schedule(dynamic) private(j,k,values) shared(grid,new_grid) reduction(+:species[:N_SPECIES+1])
for (i = 0; i < n_cells; i++) {
    for (j = 0; j < n_cells; j++) {
        for (k = 0; k < n_cells; k++) {
            values = compute_neighbors(grid, n_cells, i, j, k, values);
            species[grid[i][j][k]]++;
            new_grid[i][j][k] = compute_value(values.live_neighbors, grid[i][j][k], values.majority);
        }
    }
}
```

Figure 5: omp instruction in simulation.

As explained this is the only *omp* command we've used at this stage, firstly we use a *collapse(2)* to merge the three loops into a smaller one based on iteration over the three variables, then as no thread will have the same cell to process, we have prevented for segmentation faults or other memory access problem. Despite the fact that all the threads should have the same computational power, we decided to use a dynamic schedule instead of a static one, since some of the system's threads may be slower for an unknown reason, and after a series of tests with the tool that was provided to us, *VTune*, we saw that by using the dynamic schedule we were able to remove some of the thread idling we were facing.

Next we focused on the privacy of the variables inside of the parallel zone to avoid bad memory usage by the threads, we had to guarantee that the threads would have a local copy for the variables *j and k* corresponding to the coordinates of the cell's that belong in each partition of the cube, and also a private copy of the *struct pair values* allocated before the loop, as talked in the prior section, to receive the output of checking the neighbors.

Then we program *OpenMP* to keep the grid and new_grid shared between the threads, since there wont be accesses to the same position by different threads in the same generation. At last, in the serial version we were counting the species frequency present in each generation, at the moment with multi-threading we could keep it shared, and implement an atomic and critical to block concurrent variable access by the threads, however the results weren't the expected using this approach, which made us turn it into a reduction implementation by creating a private local variable for the species counter that when threads finish the parallel zone will sum all their respective species counter resulting in the frequency for that generation.
This final version was the one where we obtained the best results considering the criteria(speedup) for this project, which led to it being our proposed solution.

4 Results and Conclusions

4.1 Execution Times and SpeedUps

To measure the execution time of our project we used the routine *omp_get_wtime* (only measuring the execution time of the simulation through the generations), and tested the program in the PC number 1 at Lab 2 RNL (6 cores, 2 threads per core). For our analysis testing sample, we've used the four input examples available at the course project main page, and measured their execution time depending on the number of threads available to *omp*, by defining the number of threads available with the command "export OMP_NUM_THREADS=X", where X is the number of threads available. Below is the table containing our execution times in seconds, organized by the number of threads and the corresponding speedup when compared to the serial version. The samples are identified by the number of cells per side of the cube. For the speedup calculations, we've used the *Amdahl's Law* speedup calculation, using the serial time, since it is faster than the time with 1 thread.

Samples	serial	1 thread	2 threads	speedup	4 threads	speedup	8 threads	speedup
64	16,8	18,7	9,9	1,7	5	3,36	3,6	4,67
128	27,4	30,4	15,9	1,72	7,9	3,47	5,4	5,07
512	107,9	118	60,5	1,78	30,5	3,54	20,1	5,37
1024	319	337	176,8	1,8	87,6	3,64	58,3	5,47

After a quick analysis on the speedups presented on the results table, we're able to see that it usually increases accordingly with the cube's size (first-to-last order on the table), leading us to believe that our *omp* instruction and approach in this stage has completed its goal due to the speedup being similar to the number of processes used. As for the test results with 8 threads, we did not achieve an expected speedup as we did for the other two. We assume that because the PC used for testing only has 6 cores, even though it can run 2 threads per core and reach the 12 threads, the number of threads no longer allows it to run one thread per core as it did with 2 and 4 threads, causing us to believe that we were not able to achieve a comparable speedup under the testing conditions. Even with this limitation, we can see an increase in the speedup value, based on this data, we conclude that we've had a good parallelization when it comes to speedup results as we achieved values around the number of threads being used and the results also seems to show positive signs regarding the scalability of our project when running with a higher number of processes.

4.2 VTune Analysis

Here is the threading analysis provided by the VTune tool taught during the lab classes. With this analysis we're able to conclude that our program is using the CPU in an efficient way.

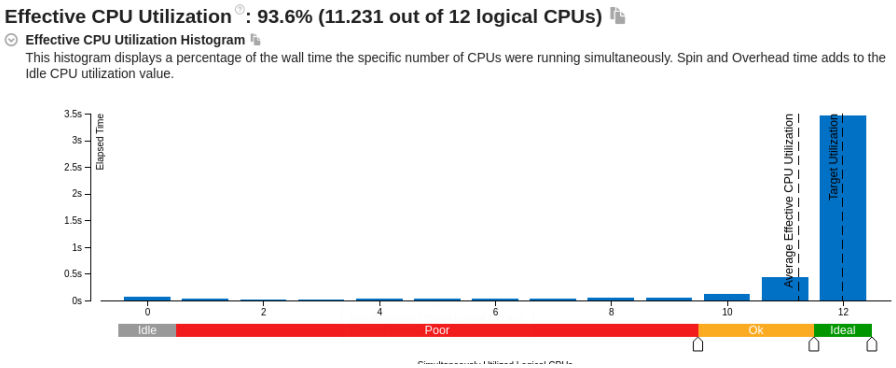

Figure 6: VTune threading analysis.