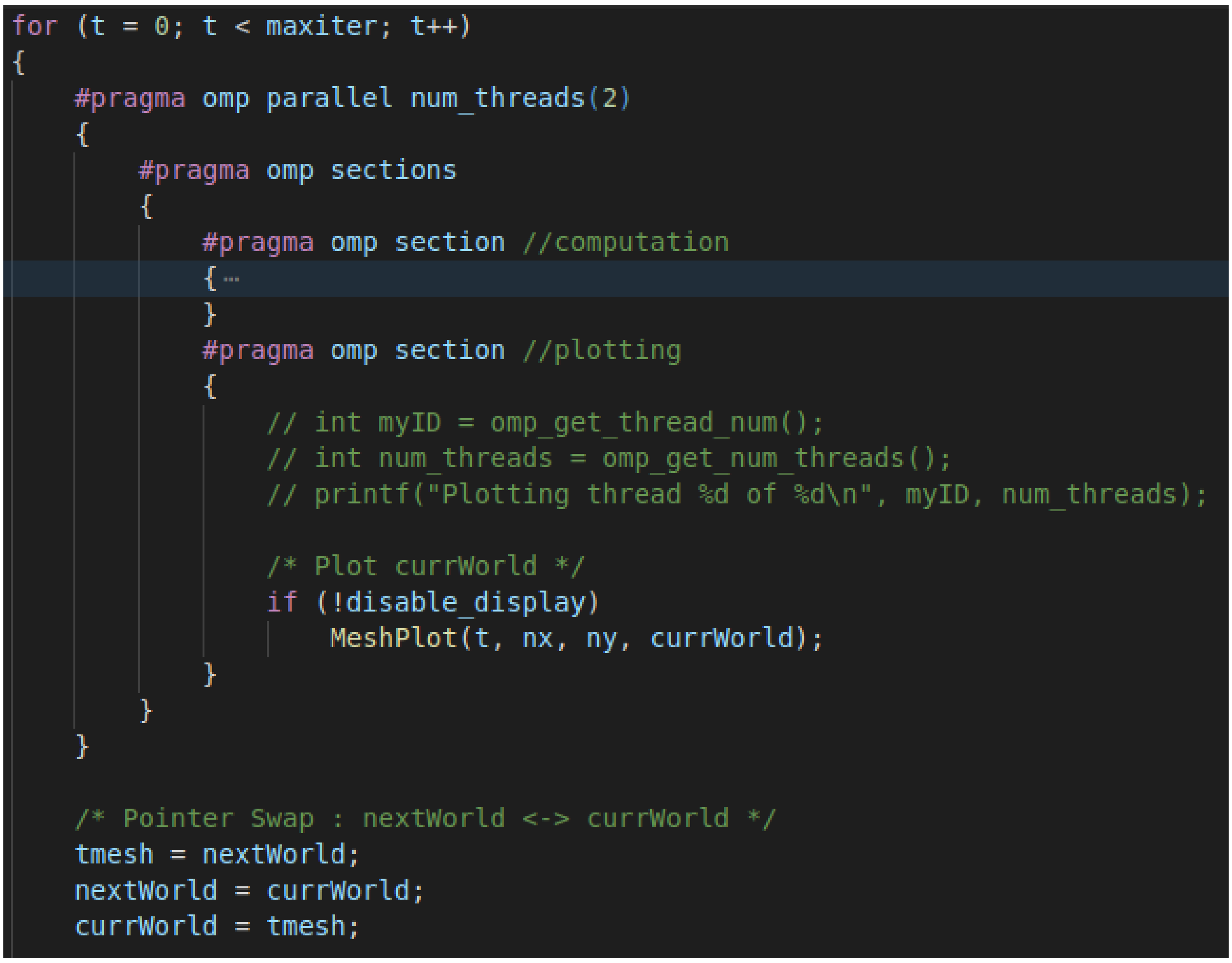
Doruk Taneli 60066

Assignment 1 Part 1 Report

Implementation

Task Parallelism

For task parallelism, I used the sections functionality of OpenMP. I created two threads, one handles the computation, the other handles the plotting. To achieve task parallelization without data dependencies, Computation part calculates nextWorld with the data from currentWorld, while Plotting part plots the currentWorld. After both of the sections finish, after the implicit barrier, the pointers of nextWorld and currentWorld are swapped.

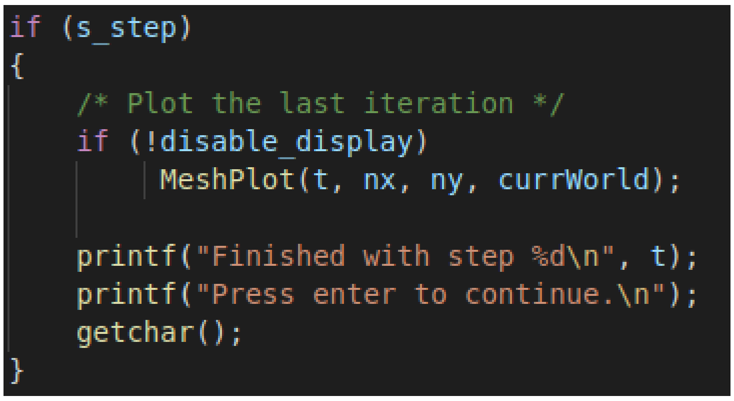


```
for (t = 0; t < maxiter; t++)
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections
        {
            #pragma omp section //computation
            {…
            }
            #pragma omp section //plotting
            {
                // int myID = omp_get_thread_num();
                // int num_threads = omp_get_num_threads();
                // printf("Plotting thread %d of %d\n", myID, num_threads);

                /* Plot currWorld */
                if (!disable_display)
                    MeshPlot(t, nx, ny, currWorld);
            }
        }
    }

    /* Pointer Swap : nextWorld <-> currWorld */
    tmesh = nextWorld;
    nextWorld = currWorld;
    currWorld = tmesh;
```

Because of these changes, the last iteration was not being plotted. To plot the last iteration, I added plotting to the last step.

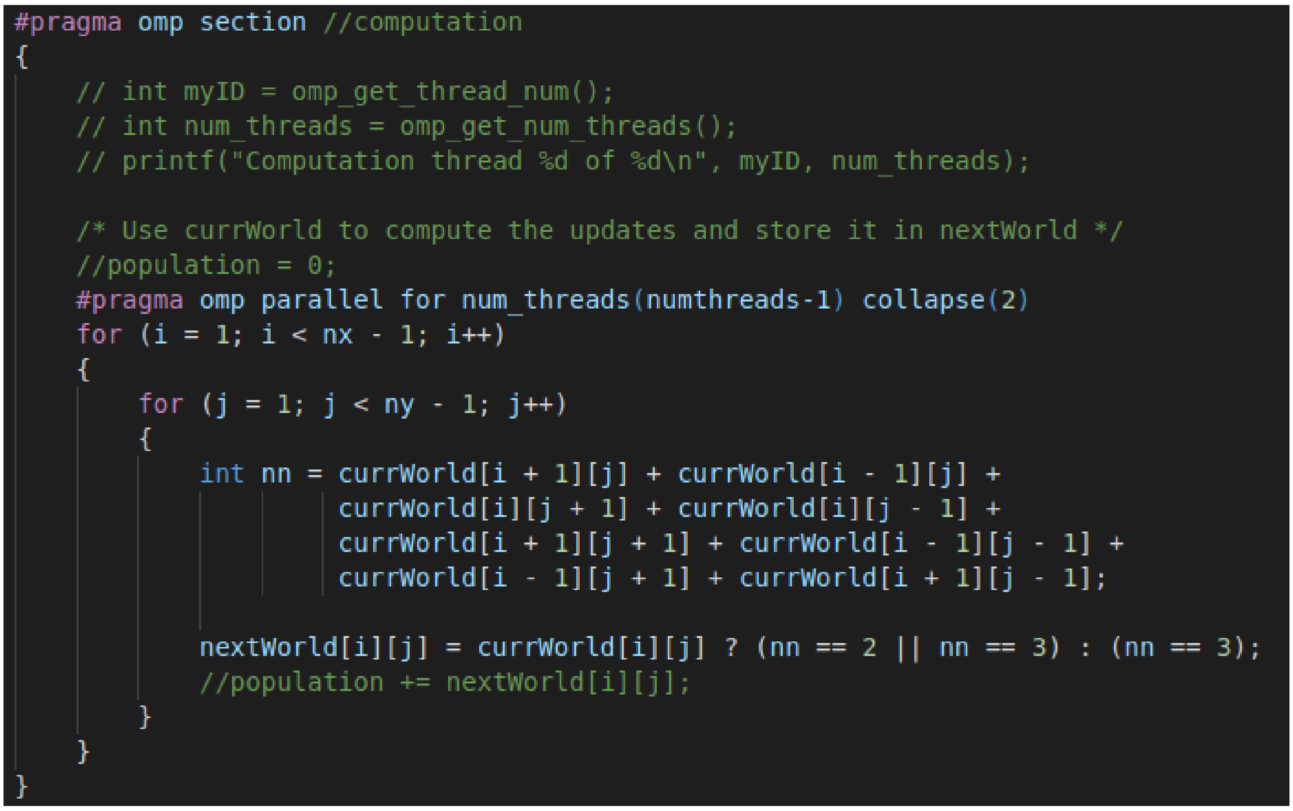


```
if (s_step)
{
    /* Plot the last iteration */
    if (!disable_display)
        MeshPlot(t, nx, ny, currWorld);

    printf("Finished with step %d\n", t);
    printf("Press enter to continue.\n");
    getchar();
}
```

Data Parallelism

For Data parallelism, I used the #pragma omp parallel functionality. I gave numthreads-1 as one of the threads was used for task parallelism. I commented out population as there were some confusion about it in the blackboard, it didn't help computations plus slowed down the program. Moreover, I used collapse(2) to parallelize the inner for loop. I tested the program with and without collapse. The version with collapse performed much better.



```
#pragma omp section //computation
{
    // int myID = omp_get_thread_num();
    // int num_threads = omp_get_num_threads();
    // printf("Computation thread %d of %d\n", myID, num_threads);

    /* Use currWorld to compute the updates and store it in nextWorld */
    //population = 0;
    #pragma omp parallel for num_threads(numthreads-1) collapse(2)
    for (i = 1; i < nx - 1; i++)
    {
        for (j = 1; j < ny - 1; j++)
        {
            int nn = currWorld[i + 1][j] + currWorld[i - 1][j] +
                     currWorld[i][j + 1] + currWorld[i][j - 1] +
                     currWorld[i + 1][j + 1] + currWorld[i - 1][j - 1] +
                     currWorld[i - 1][j + 1] + currWorld[i + 1][j - 1];

            nextWorld[i][j] = currWorld[i][j] ? (nn == 2 || nn == 3) : (nn == 3);
            //population += nextWorld[i][j];
        }
    }
}
```

Doruk Taneli 60066

Performance

Thread Count

Here is the execution times in seconds from kuacc for the following command:

```
./life -n 2000 -i 500 -p 0.2 -d -t <num-of-threads> -s 1582889460
```

Serial	33.520624
2	33.466047
4	11.360433
8	5.828014
16	4.438875
32	9.167335

Using more cores increase performance until 16 threads, after that using more cores hurts performance, which was in accordance to what Aditya got in his own trials he shared via mail. There were some variations in execution times of each submitted job, so I ran the commands with fixed seed to have a better idea of performance.

I found out that I got significantly different execution times even with same seed. For example, 32 thread execution ranged from 4s to 9s for different trials for same seed. I am not really sure of the reason.

Input Size

Here are the execution times in seconds from kuacc for the other command:

```
./life -n <input-sizes> -i 500 -p 0.2 -d -t 16 -s 1582889460
```

2000	4.362954
4000	17.007173
6000	41.514652
8000	75.652320
1000	116.194621

As we can see, the execution time is $O(n^2)$, as it should be in constant thread count and two nested for loops to compute.

Doruk Taneli 60066

Other Findings

The clear bottleneck of the program is I/O. In my local computer, running the code with plotting 2 threads took around 25 seconds, whereas when I disabled plotting, it took around 1 second.

I first tried to handle partitioning of the data parallelism for variable number of threads by myself, and encountered many problems and errors. Then I used the “omp parallel for” pragma, and realized the power and usefulness of OpenMP.