National Technical University of Athens
Electrical and Computer Engineering
Parallel Processing Systems
Semester Assignment 2023-2024

1. Familiarization with the programming environment

Team : parlab03
Ioannis Palaios - el18184

I. Introduction

We were assigned to develop a parallel version of Conway's Game Of Life assuming a shared address space architecture with OpenMP and given a serial implementation of the problem written in C. The goal is to measure the performance of the parallel version of Conway's Game Of Life using 1,2,4,6 or 8 cores of one single node (on the lab's 'Clones' cluster) and for 3 different board sizes : 64 x 64, 1024 x 1024 and 4096 x 4096. Each test case will run for 1000 generations.

II. Reviewing the given code

In Conway's Game of Life, the compute intensive part is calculating and updating the state in each cell of a 2D Tile and repeating this step for the given number of generations/time-steps (1000 in our case). The state of a cell is only dependent on its own and its neighbours previous state.

In Listing 1, we observe that it is essential to parallelize the triple nested for-loops. Due to the data accesses happening, every repetition of the outer for-loop has a direct dependence on it's previous repetition and thus making the parallelization of the outer for-loop non trivial. As a result, the obvious way to parallelize the compute intensive part is to effectively parallelize the computation happening in the two inside for-loops (B and C) into multiple CPU cores (in this assignment we will use the terms cores and threads interchangeably because we will always use the same number of threads and cores for every test case). We can separate the workload of those two for-loops to many parallel tasks where there is very limited dependencies and communication between them and thus the workload can be classified as Embarrassingly parallel.

```
int N = ARRAY_DIMENSIONS;  //array dimensions
int T = TIME_STEPS; //time steps
int ** current, ** previous;    //arrays - one for
    current timestep, one for previous timestep
int ** swap;                    //array pointer
int t, i, j, nbrs;              //helper variables

/*Allocate and initialize matrices*/
current = allocate_array(N); //allocate array for
    current time step
previous = allocate_array(N); //allocate array for
    previous time step

// for-loop A
for ( t = 0 ; t < T ; t++ ) {
    // for-loop B
    for ( i = 1 ; i < N-1 ; i++ ) {
        // for-loop C
        for ( j = 1 ; j < N-1 ; j++ ) {
                nbrs = previous[i+1][j+1] \
                       + previous[i+1][j] \
                       + previous[i+1][j-1] \
                       + previous[i][j-1] \
                       + previous[i][j+1] \
                       + previous[i-1][j-1] \
                       + previous[i-1][j] \
                       + previous[i-1][j+1];
                if ( nbrs == 3 \
                    || ( previous[i][j]+nbrs ==3 ) )
                    current[i][j]=1;
                else
                    current[i][j]=0;
            }
    }
    //Swap current array with previous array
    swap=current;
    current=previous;
    previous=swap;
}
```

Listing 1. Conway's Game of Life compute intensive part of the given serial implementation in C.

III. OpenMp parallelization

In order to effectively parallelize the serial implementation, we have to carefully select and use the appropriate OpenMP directives and constructs. Of course, we will use the basic parallel loop construct shown in Listing 2.

```
#pragma omp parallel for [clause[ [,] clause] ... ]
```

Listing 2. Basic parallel for loop contruct.

If we don't add additional clauses to this construct, a lot of customizable features like the number of threads, the shared or private variables etc. will be assigned values accordingly with OpenMP's default implementations. While, OpenMP's default implementations are useful to know, the chances are that some values have to be modified explicitly in order to maximize performance. The ones that are useful to be explicitly modified are :

- The number of threads to be created, via the *num_threads(integer-expression)* clause, assigning to it the values 1,2,4,6,8 depending on the cores we have available for each test case. This clause can be omitted because it is the same as the default OpenMP implementation.
- The private via the *private(list)* clause, assigning to them which variables should be private for each thread. In order to avoid race conditions and constant locking, we

will declare the *"j"* and the *"nbrs"* integer variables as private variables of each threads because they are declared but not initialized outside the parallel region, while each thread does not need to access these two variables of the other threads. As a result, each thread has it's own version/copy of these two variables and thus reducing significantly the cost of writing and reading them. It is important to note, that these two variables are set to null when entering the parallel region and are then initialized differently for each thread.

- The shared variables for each threads, stored in main memory, via the *shared(list)* clause. The variable that must be declared as shared between the threads is the *"previous"* array, because in each repetition of the B and C for-loops (the middle and inner one) is read-only and as a result, locks and race conditions are impossible to happen. This clause can be omitted because it is the same as OpenMP's default behaviour (i.e. each variable defined outside of the parallel region is declared as shared).

As a result, the final parallel code of the compute intensive part is shown in Listing 3.

```
int N = ARRAY_DIMENSIONS;  //array dimensions
int T = TIME_STEPS; //time steps
int ** current, ** previous;    //arrays - one for current timestep, one for previous timestep
int ** swap;                    //array pointer
int t, i, j, nbrs;              //helper variables

/*Allocate and initialize matrices*/
current = allocate_array(N); //allocate array for current time step
previous = allocate_array(N); //allocate array for previous time step

int number_of_threads = omp_get_max_threads(); //the value of the enviroment variable OMP_NUM_THREADS

// for-loop A
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for num_threads(number_of_threads) private(nbrs, j)
    // for-loop B
    for ( i = 1 ; i < N-1 ; i++ ) {
        // for-loop C
        for ( j = 1 ; j < N-1 ; j++ ) {
                nbrs = previous[i+1][j+1] \
                       + previous[i+1][j] \
                       + previous[i+1][j-1] \
                       + previous[i][j-1] \
                       + previous[i][j+1] \
                       + previous[i-1][j-1] \
                       + previous[i-1][j] \
                       + previous[i-1][j+1];
                if ( nbrs == 3 \
                    || ( previous[i][j]+nbrs ==3 ) )
                    current[i][j]=1;
                else
                    current[i][j]=0;
            }
    }
    //Swap current array with previous array
    swap=current;
    current=previous;
    previous=swap;
}
```

Listing 3. Conway's Game of Life compute intensive part of the parallel implementation in C with OpenMP.

We chose to parallelize the for-loop B instead of the for-loop C. This choice was made in order to avoid excessive thread creation and management. If the for-loop C was parallelized, there would be the creation of N-1 more threads than if we had parallelized the for-loop B and then the overhead of thread creation and management would be significantly bigger and no discernibly better distribution of the workload would be observed. As a result, we anticipate that in small dimensions, such as 64x64, thread number increase would maybe reduce performance due to thread management overhead.

IV. BENCHMARKING THE PARALLEL PROGRAM

As previously mentioned, we want to benchmark the program for 5 different core numbers (1,2,4,6 and 8 - the value of the *'number_of_threads'* variable) and 3 different board sizes (64 x 64, 1024 x 1024 and 4096 x 4096 - the value of the *'N'* variable in Listing 3). In order to automate the process of running all these different test cases, we created a bash script shown in Listing 4.

```
#!/bin/bash

cores=(1 2 4 6 8)
board_sizes=(64 1024 4096)
for number_of_cores in ${cores[@]}; do
  for board_size in ${board_sizes[@]}; do
        #PBS -N run_omp_Game_Of_Life_${number_of_cores}_cores_${board_size}

        ## Output and error files
        #PBS -o run_omp_Game_Of_Life_${number_of_cores}_cores_${board_size}.out
        #PBS -e run_omp_Game_Of_Life_${number_of_cores}_cores_${board_size}.err

        ## How many machines should we get?
        #PBS -l nodes=1:ppn=$number_of_cores

        ##How long should the job run for?
        #PBS -l walltime=00:10:00

        ## Start
        ## Run make in the src folder (modify properly)

        module load openmp
        cd /home/parallel/parlab03/parlab-ex01
        export OMP_NUM_THREADS=$number_of_cores
        echo "$OMP_NUM_THREADS"
        ./Game_Of_Life $board_size 1000
  done
done
```

Listing 4. The bash script used in order to automate running test case for different numbers of cores and board_sizes

After running the script, we get the total execution time for every different combination of core number and board size (i.e. 15 combinations). The plot showing the execution times for each different combination in logarithmic scale is shown on Figure 1.

We observe that there is a significant increase in performance as we increase the thread number between 1, 2 and 4 threads. This is because in these numbers of threads the job/process is compute-bound which means that the time it takes for it to complete is determined principally by the speed of the central processor and as a result CPU utilization

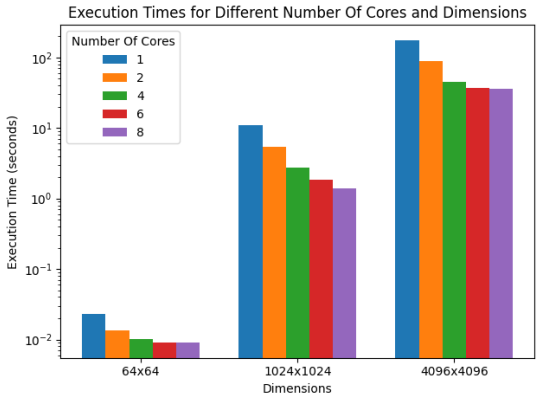

Figura 1. Execution times of Conway's Game Of Life parallel OpenMP implementation in logarithmic scale parallelizing the middle for-loop

is high. This is visible for all the dimensions, while the performance increase in more visible in the biggest dimensions (i.e. 4096x4096).

But as we increase the number of threads to 6 or 8 we notice that the performance increase is not as significant as before (or maybe negligible between them, achieving little to no scaling in execution times). This is possibly due to that the job/process is I/O Bound which means that the time to complete a computation is mainly determined by the period spent waiting for the input and output of the data (i.e. more time is spent requesting the data rather than processing it).

Additionally, there is no significant difference between the different dimensions, because the computation and I/O overhead is significantly bigger than the thread management overhead in all of the tested dimensions and thus, the proportion between them stays relatively the same (and in favor of the first type). As a result, similar behaviour is observed between the different dimensions.

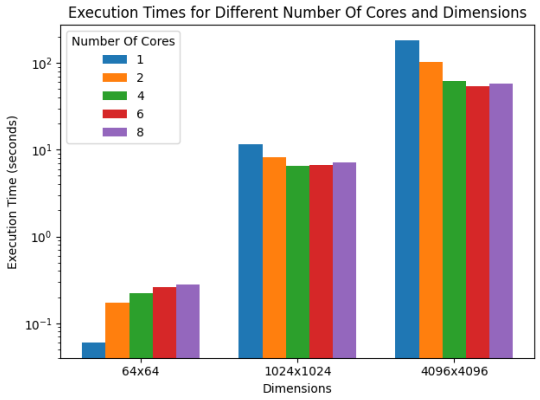

Figura 2. Execution times of Conway's Game Of Life parallel OpenMP implementation in logarithmic scale parallelizing the inner for-loop

We also, tested and plotted the execution times for the same job/task but parallelized the inner for loop, instead of the middle one. The results are shown in Figure 2. As we expected, in the relatively small dimensions where the proportion between Computation and I/O overhead compared to the thread management overhead might be in favor of the latter (i.e. 64x64 dimensions), we observe that the increase of thread size decreases performance.

A similar phenomenon is visible in the two higher dimensions, where the execution times with a higher thread number (i.e. 4,6 or 8) are smaller than those with a small thread number (i.e. 1 or 2) but they cannot further decrease as we increase the thread number over 4, likely because of the higher thread management overhead and the I/O intensive workload.

Obviously, the performance gets significantly better when we parallelize the middle for-loop instead of the inner one, for the thread sizes bigger than one. This is happening due to the reasons we mentioned above (i.e. more threads are created when parallelizing the inner for loop so the thread management overhead increases).

V. CONCLUSION

In this assignment, we effectively parallelized and benchmarked a seemingly Embarrassingly Parallel problem. We benefited from the abstractions that OpenMP's API provides and parallelized a double nested for loop. Additionally, we shown the differences between the different thread sizes and dimensions chosen for each test case, while also compared the parallelization between the middle and inner for loops. Furthermore, there is a lot of room for improvement and optimizations via OpenMP's API which we will explore in the future.

REFERENCIAS

[1] OpenMP Application Program Interface Version 4.0 - July 2013 https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.
[2] OpenMP Programming Book. https://passlab.github.io/OpenMPProgrammingBook/.