Report on task 3 (template)

**Name: Alua Onayeva**

**Group: BD-2008**

**E-mail: 201260@astanait.edu.kz**

**Main part:**

Step 0:

We will be using the sequential ray tracing program from Task 1. Download and install Mini-Rt library (https://github.com/georgy-schukin/mini-rt), if necessary.

Step 1: Prepare a directory for the Task 3

In your personal directory:

* Create directory “Task 3”
* Copy the sequential program to this new directory
* Rename the file to raytracing\_openmp.cpp

Step 2: Implement parallel program with OpenMP

Use OpenMP to parallelize the sequential ray tracing program (edit raytracing\_openmp.cpp); the single image should be computed in parallel by many threads. Use **#pragma omp parallel** and **#pragma omp for** directives (or **#pragma omp parallel for** combined directive) to parallelize the main computational loop (in which image is computed pixel by pixel).

*Hint*: you can use [this program template](https://github.com/georgy-schukin/hpc-course/blob/master/task_templates/task3/raytracing_openmp.cpp) as a starting point.

*Hint*: study [this program example](https://github.com/georgy-schukin/hpc-course/blob/master/examples/openmp/array_for/array_for.cpp) about parallelizing ‘for’ loop with OpenMP.

To compile your program with OpenMP support (Linux/Mac):

g++ -O3 -fopenmp -o raytracing\_openmp raytracing\_opnmp.cpp -lminirt

For other compilers and operating systems: look in documentation how to enable OpenMP.

To run your OpenMP program with N threads, first set value of OMP\_NUM\_THREADS environment variable:

export OMP\_NUM\_THREADS=*N* && ./raytracing\_openmp <args>

Or set number of threads with omp\_set\_num\_threads() function or num\_threads() clause (setting number of threads this way will override value from OMP\_NUM\_THREADS variable).

Compare performance with different parameters of the **schedule** clause for ‘for’ directive (for example, **schedule(static)**, **schedule(static, 1)** and **schedule(dynamic)**). Don’t forget to recompile the program after changing the parameters. Explain the results. Why do some parameters provide better performance? Why are the others worse?

Step 3: Study performance of your parallel program

1. Use omp\_get\_wtime() to measure the execution time for the main loop:

**double start = omp\_get\_wtime();**

#pragma omp …

for(int x = …)

for(int y = …) {

    const auto color = viewPlane.computePixel(

scene, x, y, numOfSamples);

…

}

**double end = omp\_get\_wtime();**

**double execution\_time = end - start;**

std::cout << “Time = “ << execution\_time << std::endl;

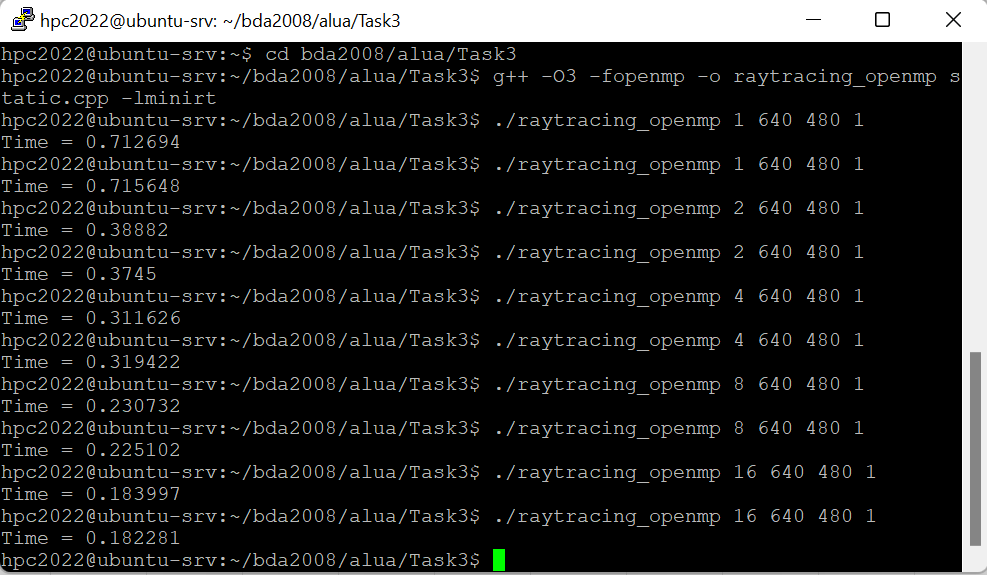
1. Select such a scene and rendering parameters (image size, number of samples, depth of recursion, etc.), that the execution time of the program, when running on 1 thread, is more than several seconds.
2. Measure the execution time for the parallel program on 1, 2, 4, 8, 16 threads. For accuracy you can do several runs (>5) on each number of threads and choose the minimal time among runs for this number of threads.

g++ -O3 -fopenmp -o raytracing\_openmp static.cpp -lminirt

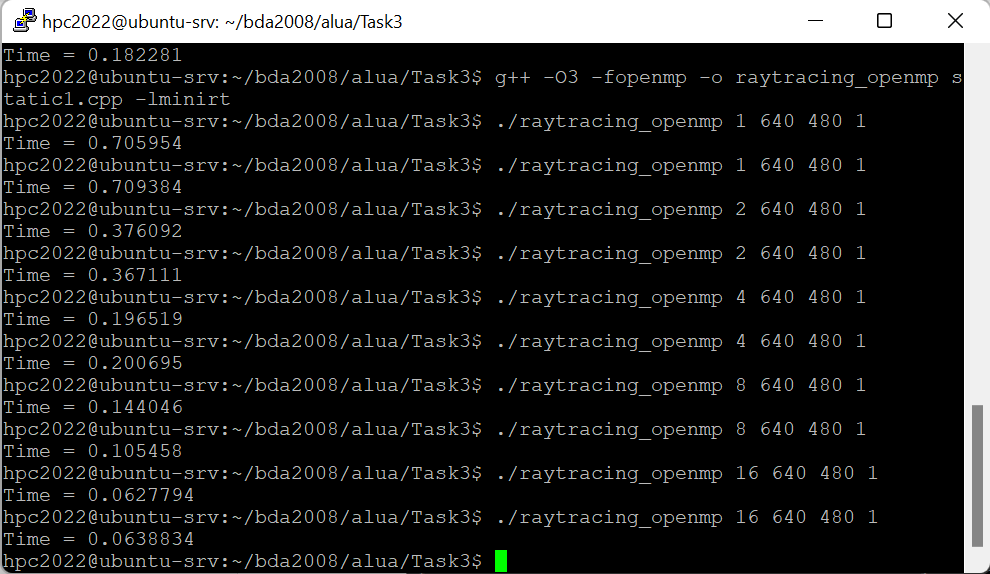
g++ -O3 -fopenmp -o raytracing\_openmp static1.cpp -lminirt

g++ -O3 -fopenmp -o raytracing\_openmp dynamic.cpp -lminirt

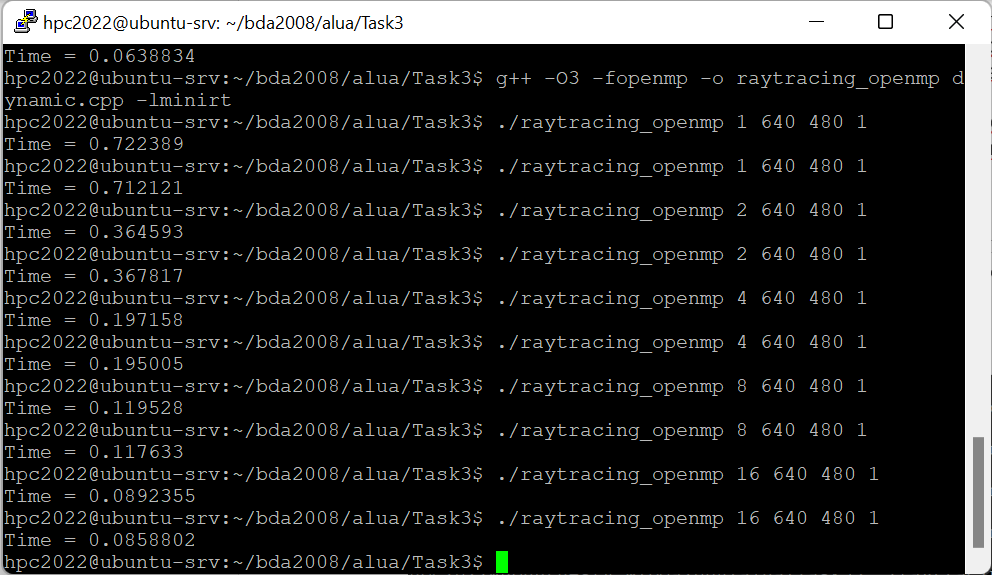
**static**



**static, 1**

****

**dynamic**

****

1. Build plots/tables for:
   1. The execution time (to demonstrate how it depends on the number of threads)
   2. Speedup: Speedup(N) = Time(1) / Time(N), N - number of threads
   3. Efficiency: Efficiency(N) = Speedup(N) / N

parameters: 640 480 1

schedule (static)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Threads number | 1 | 2 | 4 | 8 | 16 |
| Execution time | 0.712 | 0.3745 | 0.3116 | 0.2251 | 0.1822 |
| Speed-up | 1 | 1.901 | 2.285 | 3.163 | 3.907 |
| Efficiency | 1 | 0.9505 | 0.5712 | 0.395 | 0.2441 |

schedule (static, 1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Threads number | 1 | 2 | 4 | 8 | 16 |
| Execution time | 0.706 | 0.367 | 0.1965 | 0.105 | 0.0627 |
| Speed-up | 1 | 1.923 | 3.593 | 6.723 | 11.256 |
| Efficiency | 1 | 0.9615 | 0.898 | 0.8403 | 0.7035 |

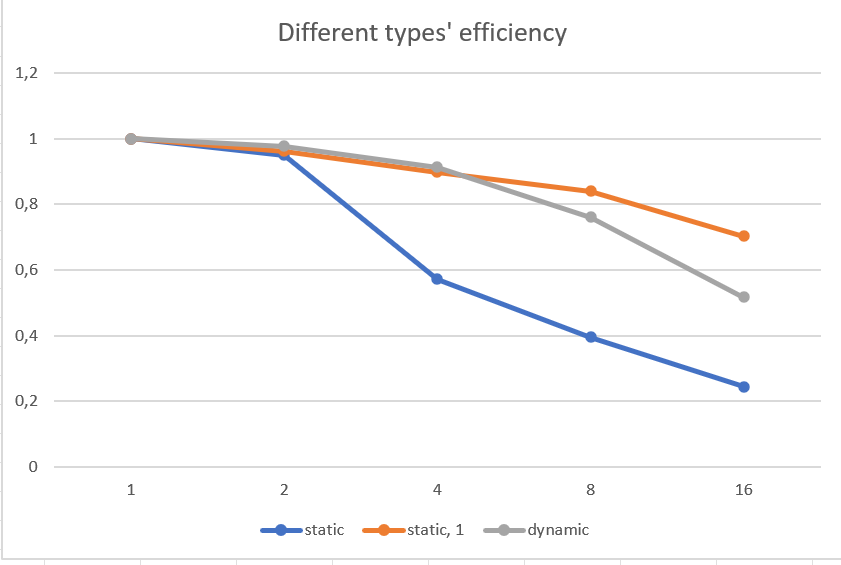
schedule (dynamic)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Threads number | 1 | 2 | 4 | 8 | 16 |
| Execution time | 0.712 | 0.364 | 0.195 | 0.117 | 0.086 |
| Speed-up | 1 | 1.956 | 3.651 | 6.085 | 8.279 |
| Efficiency | 1 | 0.978 | 0.913 | 0.761 | 0.517 |

Remember that you have to compare performance of the program with different parameters of the **schedule** clause. So, you will have multiple lines on your plots for different versions of parameters.

Explain reasons for the (possible) difference in the performance of the program with different **schedule** parameters (you may use other parameters than in the example here).

**In my case graph will look like this:**



As we can see from the graph, schedule(static, 1) shows the best performance.

The second one is schedule(dynamic), and the last one is schedule(static).

Step 4: Commit and push your changes to the Gitlab server

Link: <https://github.com/loopiiu/hpc_task3.git>

Step 5: Conclusion in a free form

Dynamic scheduling is better in cases when the amount of time for iterations differs. While static is better when the amount of time for iteration is approximately the same.

Number of iterations that thread take is called chunk. Scheduling became more static when the chunk size increase and became dynamic when the chunk size decrease.

As we can see from the graph above, our static scheduling performs better, which indicates that our time spend for each iteration is approximately the same.

Piece of code that is changed:

#pragma omp parallel

    {

        #pragma omp for schedule(static)

        for(int x = 0; x < viewPlaneResolutionX; x++)

        {

            for(int y = 0; y < viewPlaneResolutionY; y++)

            {

                const auto color = viewPlane.computePixel(scene, x, y, numOfSamples);

                image.set(x, y, color);

            }

        }

    }