Report on task 3

**Name: Akzhurek Asakrbek**

**Group: BD-2006**

**E-mail: 201457@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);

…

}

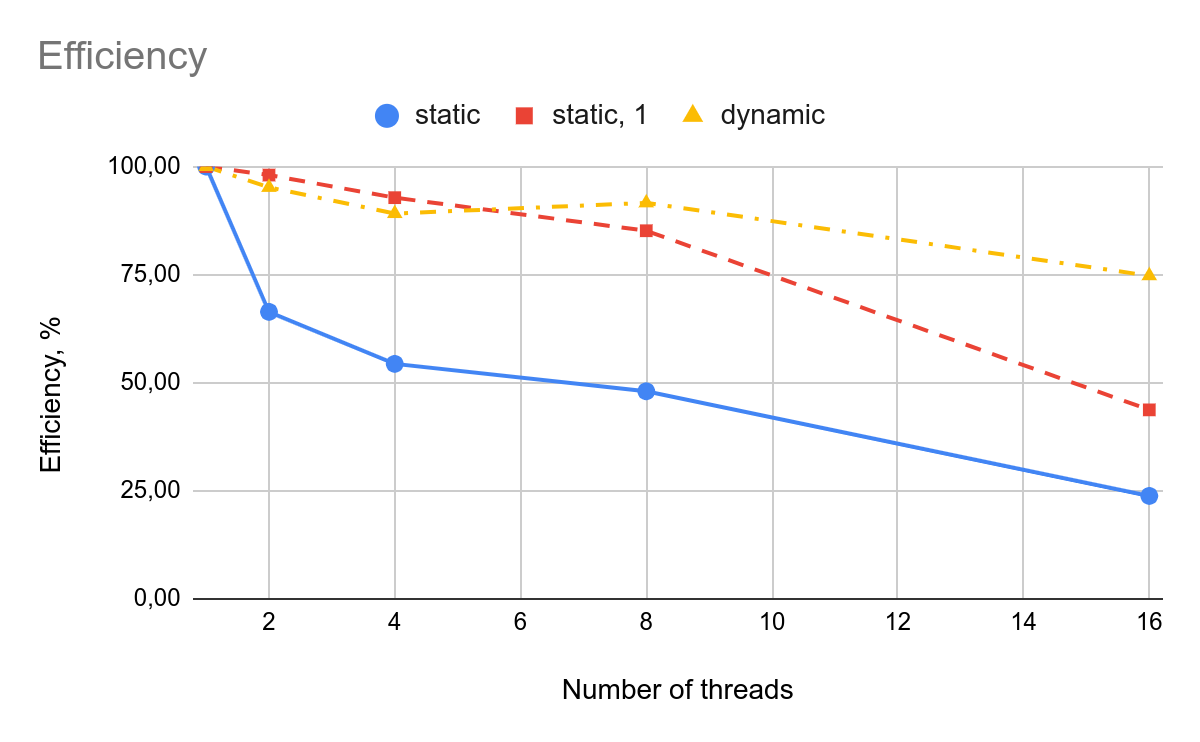
**double end = omp\_get\_wtime();**

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

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

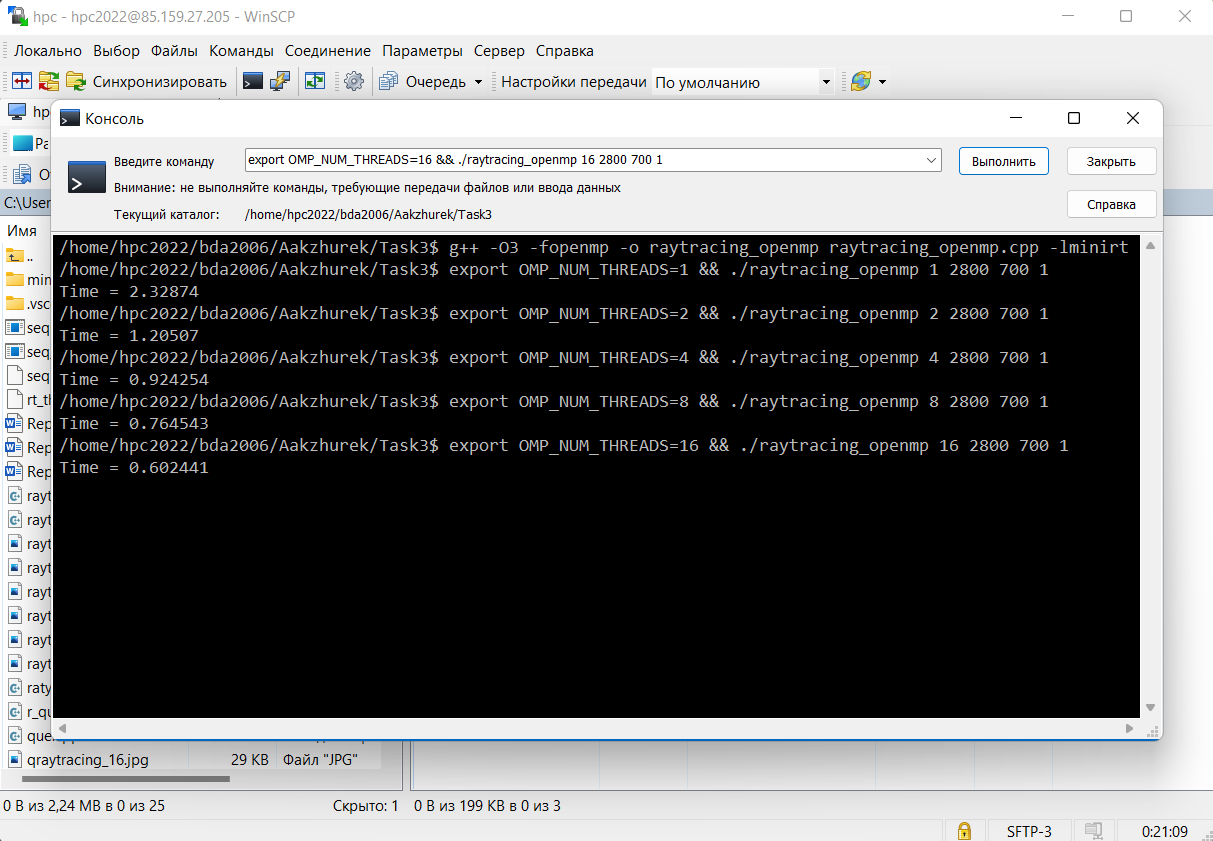
1. Select such a scene and rendering parameters (image size, number of samples, depth of recursion, etc.), that the execution time of the program, when running on 1 thread, is more than several seconds.
2. Measure the execution time for the parallel program on 1, 2, 4, 8, 16 threads. For accuracy you can do several runs (>5) on each number of threads and choose the minimal time among runs for this number of threads.
3. 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

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. For example:

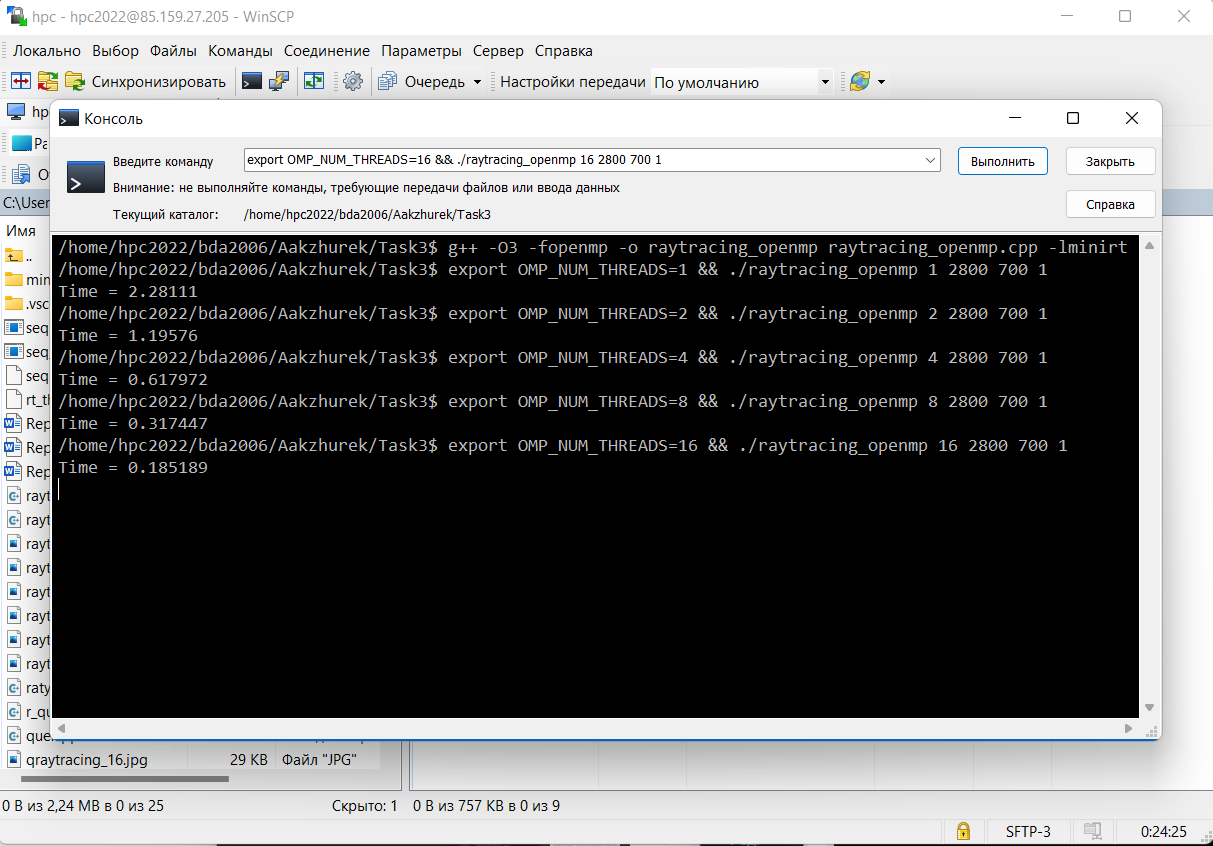


Explain reasons for the (possible) difference in the performance of the program with different **schedule** parameters (you may use other parameters than in the example here).

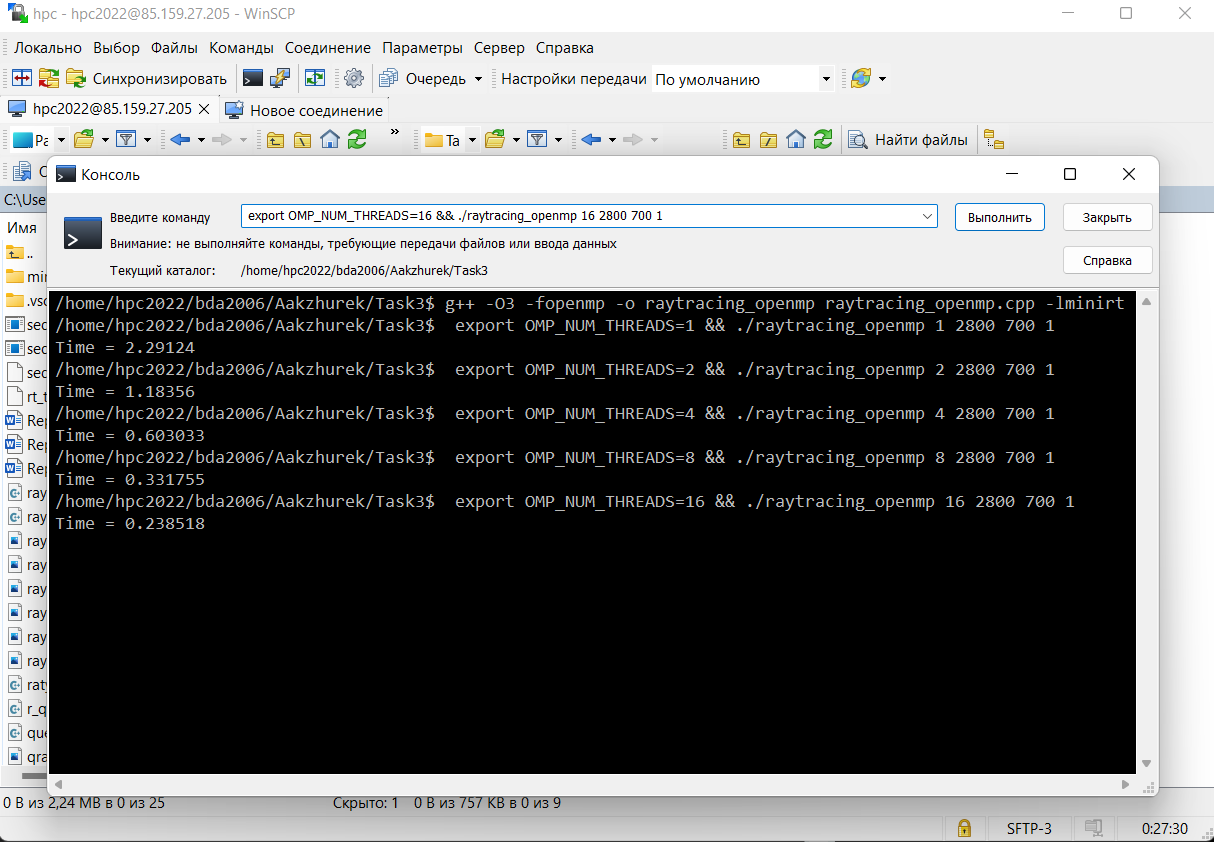
**Static**

****

**Static+1**

****

**Dynamic**

****

**Static T1=2.32874**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Threads | 2 | 4 | 8 | 16 |
| Time | 1.20507 | 0.924254 | 0.764543 | 0.602441 |
| Speedup | 1.93245206 | 2.51958877 | 3.04592417 | 3.86550716 |
| Efficiency | 0.96622603 | 0.62989719 | 0.38074052 | 0.2415942 |

**Static+1 T1=2.28111**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Threads | 2 | 4 | 8 | 16 |
| Time | 1.19576 | 0.617972 | 0.317447 | 0.185189 |
| Speedup | 1.90766542 | 3.69128375 | 7.18579794 | 12.3177403 |
| Efficiency | 0.95383271 | 0.92282094 | 0.89822474 | 0.76985877 |

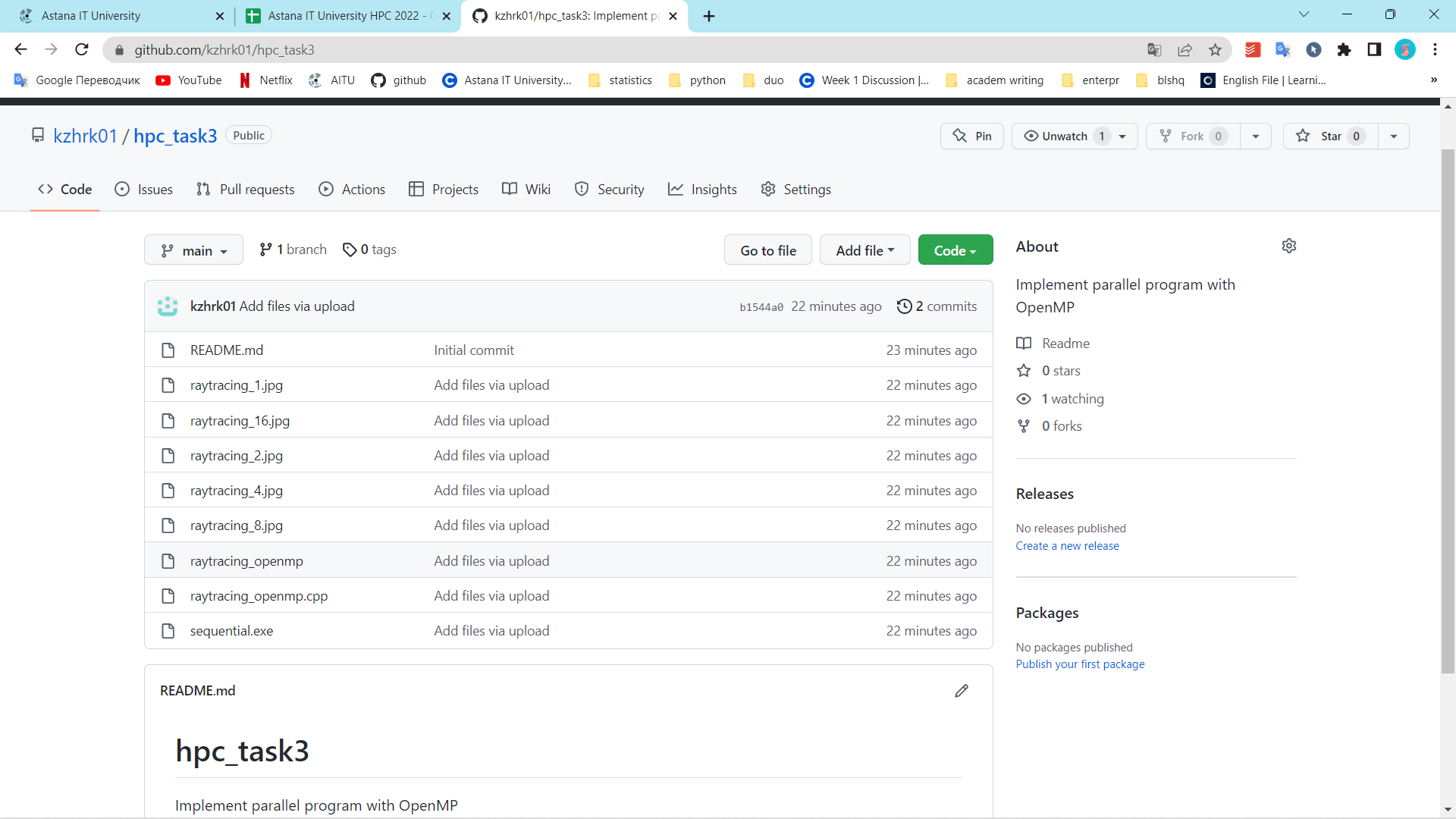
**Dynamic T1=2.29124**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Threads | 2 | 4 | 8 | 16 |
| Time | 1.18356 | 0.603033 | 0.331755 | 0.238518 |
| Speedup | 1.93588834 | 3.79952673 | 6.90642191 | 9.6061513 |
| Efficiency | 0.96794417 | 0.94988168 | 0.86330274 | 0.60038446 |

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

Upload your source code files in Task 3 directory to your Github repository, include a link to it in the report.

<https://github.com/kzhrk01/hpc_task3.git>



Step 5: Conclusion in a free form

The findings above show that static+1 gives itself in the greatest way, whereas static provides itself in the worst way.