Report on task 2 (template)

**Name: Sarsenov Damir**

**Group: BDA-2008**

**E-mail: immbishe@gmail.com**

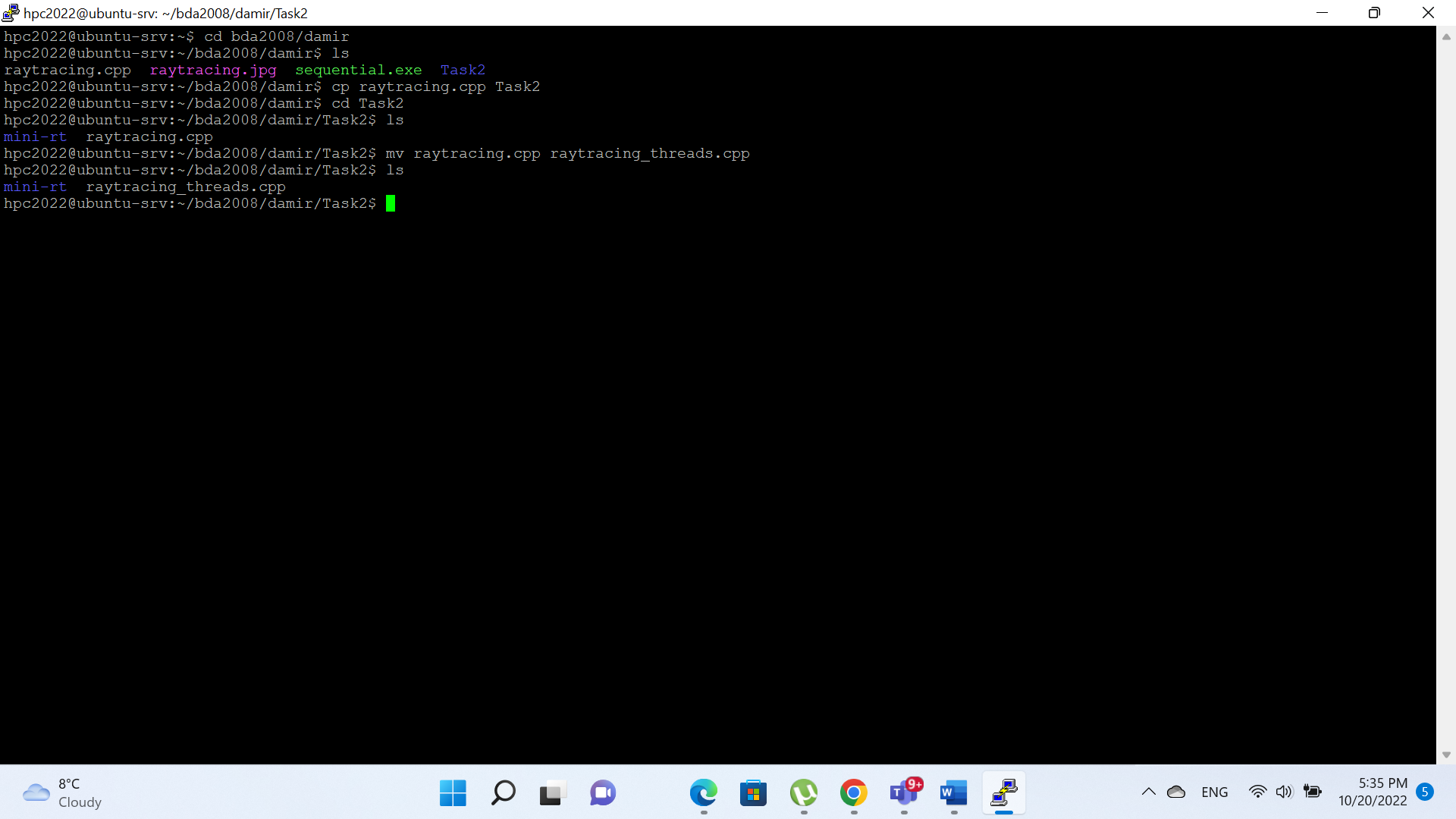
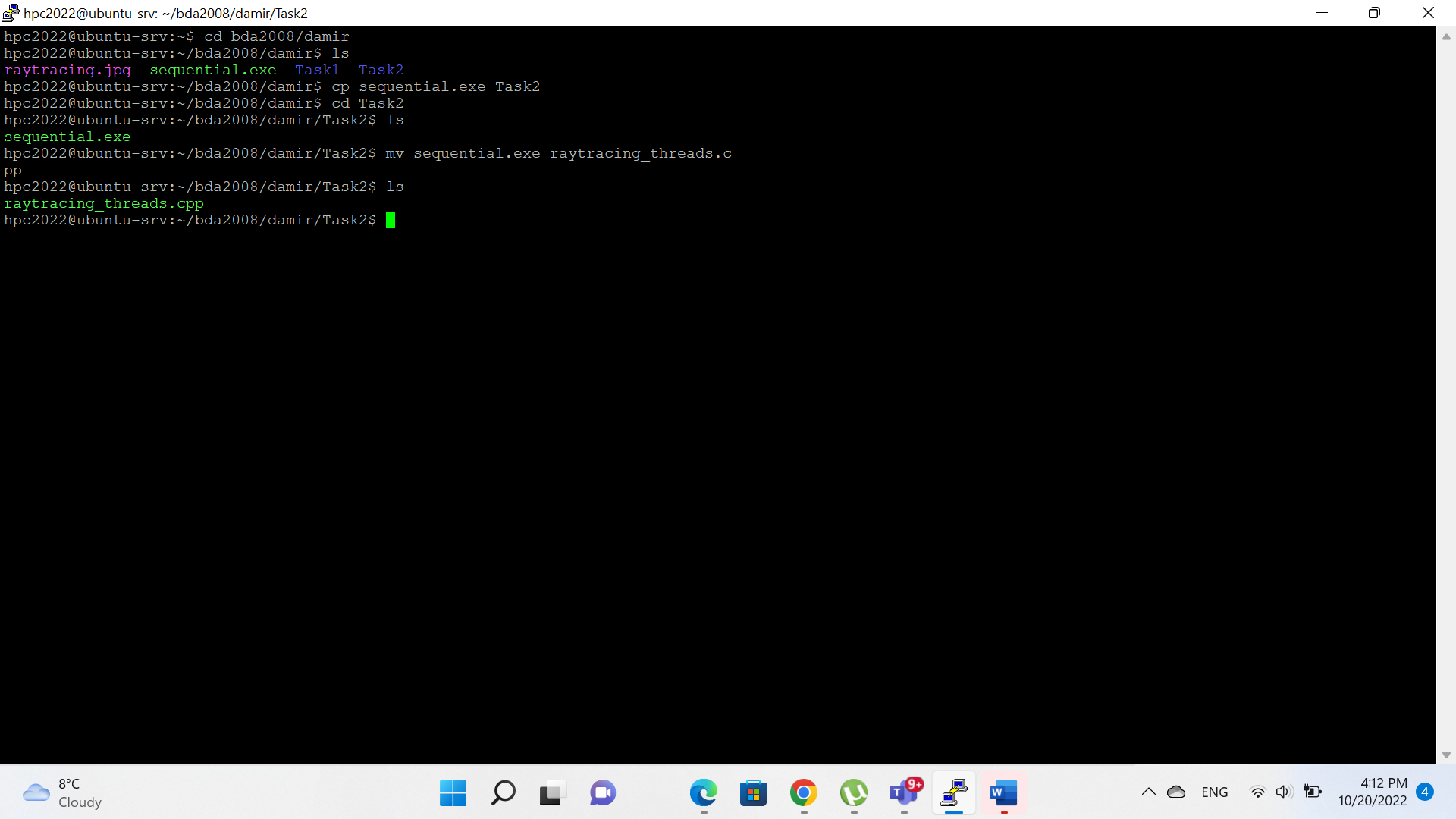
**Main part:**

# 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 2

In your personal directory:

* Create directory “Task 2”
* Copy sequential program to this new directory
* Rename the file to raytracing\_threads.cpp
* 
* 
* Text

  Description automatically generatedA screenshot of a computer

  Description automatically generated with medium confidence

# Step 2: Implement static scheduling with POSIX/C++ threads

Use POSIX/C++ Threads to parallelize the sequential ray tracing program (edit raytracing\_threads.cpp); the single image should be computed in parallel by many threads.

Partition the computation of the whole image onto blocks of pixels of equal size (you can divide image by rows or by columns), create threads and make different threads compute different blocks (one block per thread).

*Hint*: you can use [this program template](https://github.com/georgy-schukin/hpc-course/blob/master/task_templates/task2/raytracing_threads.cpp) as a starting point.

*Hint*: study [this program example](https://github.com/georgy-schukin/hpc-course/blob/master/examples/threads/cpp_array/cpp_array.cpp) about array computation and static scheduling.

# Step 3: Study performance of your parallel program

1. Because now actual code to compute pixels may be moved into tasks and threads, we will take time between when all threads are created/started and all threads are done as execution time of the program.   
     
   Use std::chrono library to measure wall time (you can also gettimeofday or another function for measuring real time):

#include <chrono>

using namespace std::chrono;

…

// init queue of tasks

**auto ts = high\_resolution\_clock::now();**// create threads

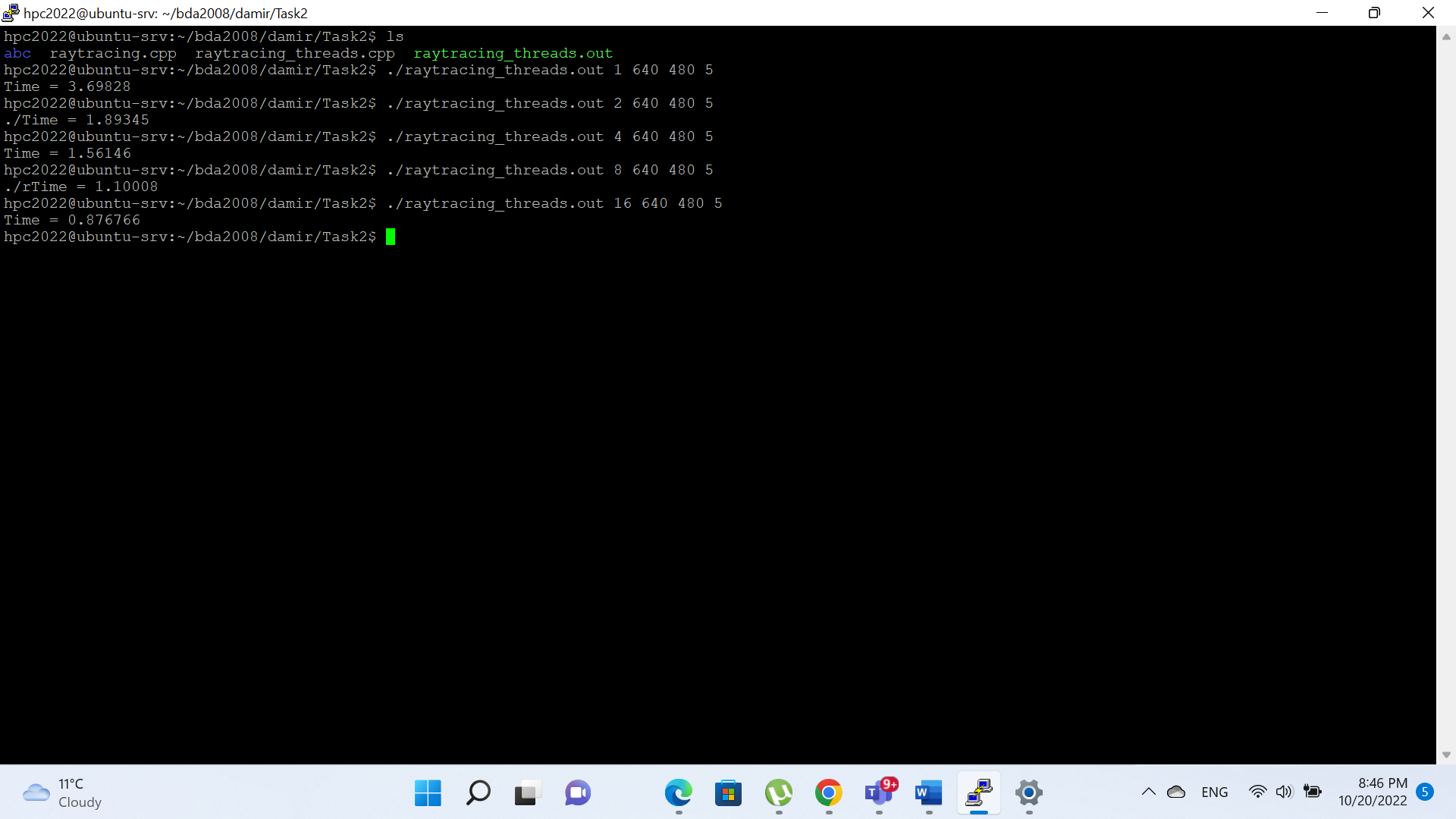
// join threads

**auto te = high\_resolution\_clock::now();**

**double execution\_time = duration<double>(te - ts).count();**

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

1. Select such a scene and rendering parameters (image size, number of samples), 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)



A screenshot of a computer

Description automatically generated

* 1. Speedup: Speedup(N) = Time(1) / Time(N), N - number of threads

1. 3.69828 ÷ 1.89345 = 1.95319
2. 3.69828 ÷ 1.56146 = 2.36847
3. 3.69828 ÷ 1.10008 = 3.36182
4. 3.69828 ÷ 0.876766 = 4.21819
   1. Efficiency: Efficiency(N) = Speedup(N) / N
5. 1.95319 ÷ 2 = 0.97659
6. 2.36847 ÷ 4 = 0.59211
7. 3.36182 ÷ 8 = 0.42022
8. 4.21819 ÷ 16 = 0.26363

# Step 4\*: Implement dynamic scheduling with POSIX/C++ threads

Partition the computation of the whole image onto a big number of tasks; each task should be to compute a small part of the image (block of pixels, row, column, etc). Create all tasks and place them into a queue (“bag of tasks”); your parallel threads will have to take tasks (pieces of work) from this shared queue one by one until all tasks are done and this way threads will complete the image together. Make sure necessary synchronization (mutex) is used to make threads work with a shared queue correctly. But also make sure that real work (computing pixels) happens in parallel outside of critical sections.

There are different ways to partition the image. For example:

* A piece of work is a 2D block of pixels; all blocks are of the same/different size (what size will be the best to optimize performance?)
* A piece of work is a row (or column) of pixels in the image
* A piece of work is a single pixel – this could be too small a piece of work and overhead (time which programs spends to manage task processing) will be too high

*Hint*: study [this program example](https://github.com/georgy-schukin/hpc-course/blob/master/examples/threads/cpp_queue/cpp_queue.cpp) about working with a shared queue.

Repeat time measurements for dynamic scheduling with the same input parameters as for static scheduling, build tables/plots for working time, speedup and efficiency, compare the results.

# Step 5: Commit and push your changes to the Gitlab server

Upload your source files (.cpp and .h) and scene files (.txt) in Task 2 directory to your Github repository, include a link to it in the report.

<make screenshots>

# Step 6: Conclusion in a free form

Explain how you partitioned the work among the threads, explain the performance evaluation results.

In conclusion, having more threads or cores allows for the completion of more tasks concurrently, but this does not necessarily translate to faster completion of those processes. The better, the more threads and cores. Your computer will run faster the more you have. This is possible because tasks can be divided and carried out concurrently. They can only use time that would otherwise be squandered to boost the computer's efficiency. This improvement can speed up some processing processes and cut down on running time. The more threads your computer has, the more efficiently it will do specialized jobs. A single process can handle a number of tasks at once when there are numerous threads.