

# Report on task 2 (template)

Name: Alua Onayeva

Group: BD-2008

E-mail: 201260@astanait.edu.kz

Main part:

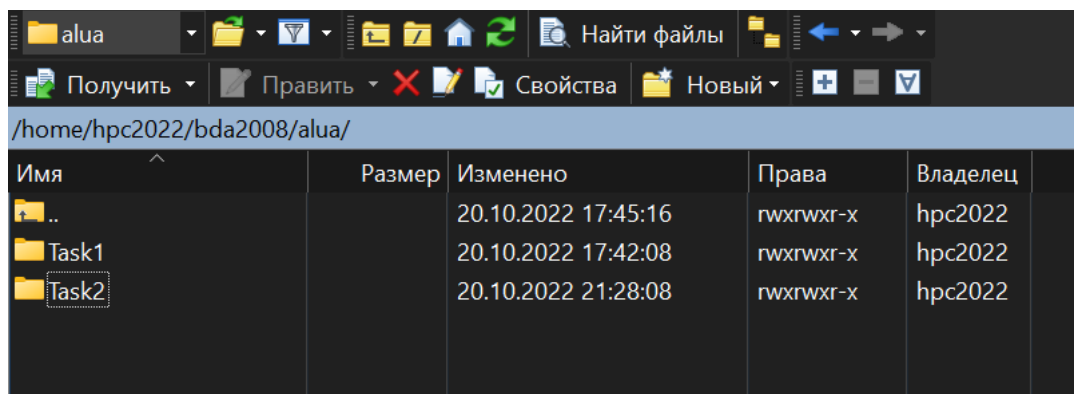
## Step 0:

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 2

In your personal directory:

- Create directory "Task 2"
- Copy sequential program to this new directory
- Rename the file to raytracing\_threads.cpp



## 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](#) as a starting point.

*Hint:* study [this program example](#) 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):

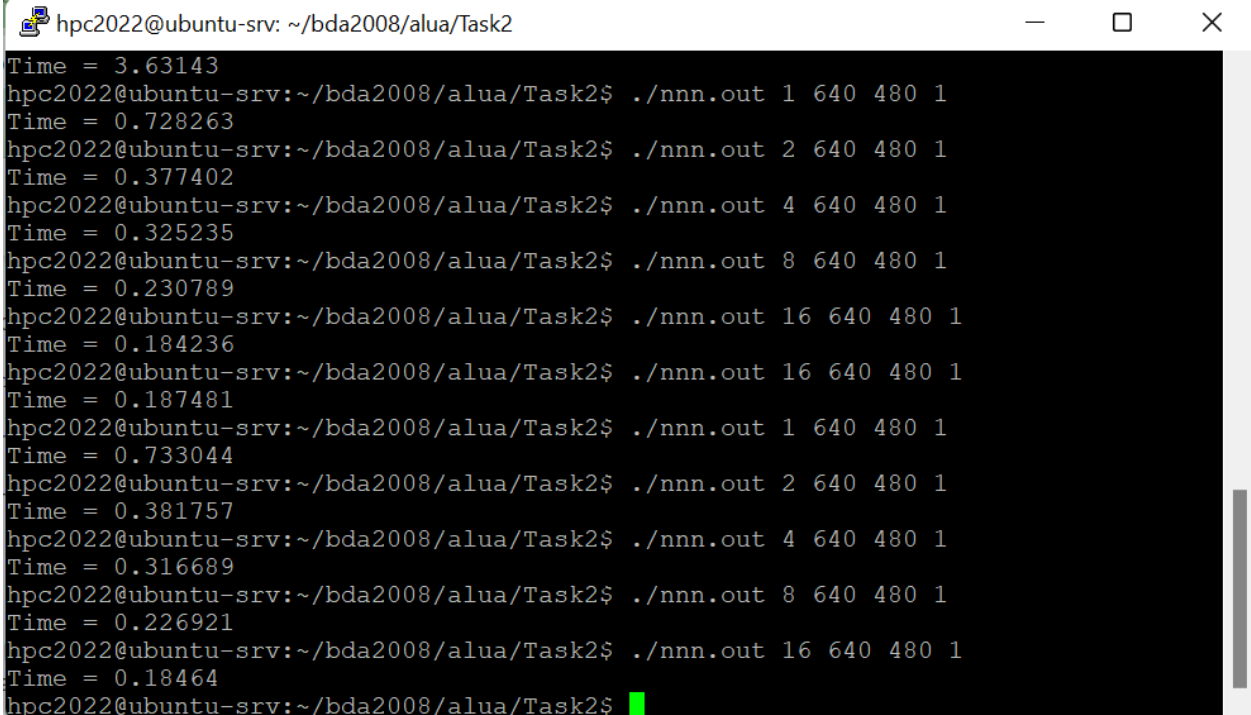
```
#include <chrono>
using namespace std::chrono;
...

// 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;
```

2. 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.
3. 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.
4. Build plots/tables for:
  - a. The execution time (to demonstrate how it depends on the number of threads)
  - b. Speedup:  $\text{Speedup}(N) = \text{Time}(1) / \text{Time}(N)$ ,  $N$  - number of threads
  - c. Efficiency:  $\text{Efficiency}(N) = \text{Speedup}(N) / N$



A terminal window titled "hpc2022@ubuntu-srv: ~/bda2008/alua/Task2" displays the execution times of a program for various thread counts. The output shows that as the number of threads increases from 1 to 16, the execution time decreases significantly, indicating parallelization. The times are as follows:

Thread Count	Execution Time (s)
1	3.63143
2	0.728263
4	0.377402
8	0.325235
16	0.230789

The terminal also shows several repeated runs for each thread count, with the times generally staying consistent, except for a slight increase at 16 threads in the second set of runs (0.184236 and 0.187481).

Running multiple time the same parameters to find the best time.

Threads number	1	2	4	8	16
Execution time	0.728	0.377	0.316	0.227	0.184
Speed-up	1	1.931	2.303	3.207	3.956
Efficiency	1	0.9655	0.575	0.401	0.247

## 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:

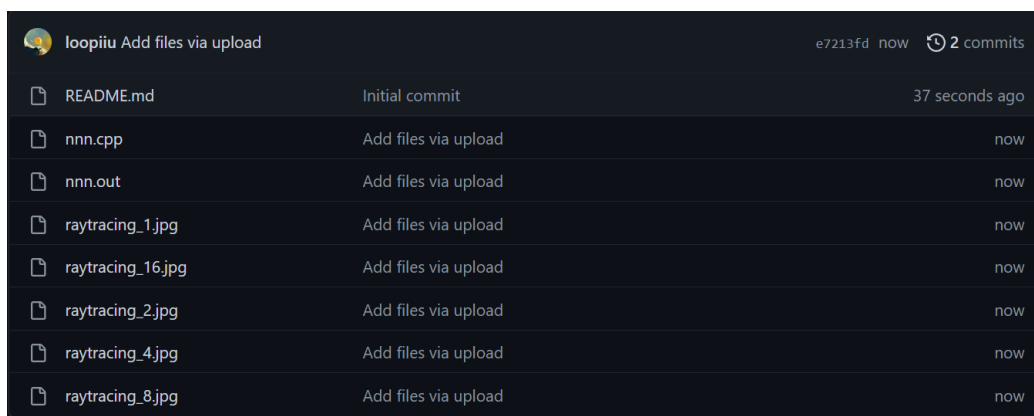
- A piece of work is a 2D block of pixels; all blocks are of the same/different size (what size will be the best to optimize performance?)
- A piece of work is a row (or column) of pixels in the image
- A piece of work is a single pixel – this could be too small a piece of work and overhead (time which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

*Hint:* study [this program example](#) 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.



[https://github.com/loopiiu/hpc\\_task2.git](https://github.com/loopiiu/hpc_task2.git)

## Step 6: Conclusion in a free form

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

Majority of the code was written in repository. We have just added several corrections.

It was said to add parameters into function. We have added two parameters into threadFunc() function, such as start and end. And added it into x loop function. Because we are dividing our .jpg file into a lot of small particles by x and y axes.

```
void threadFunc(Scene &scene, ViewPlane &viewPlane, Image &image, int start, int end, int sizeY, int numOfSamples)
```

Also defined block as division of 'viewPlaneResolutionX' to 'numberOfThreads'.

Each thread will be responsible for calculating different part of the images.

In order to pass data into each thread multiply the block size into I, as there is an iteration.

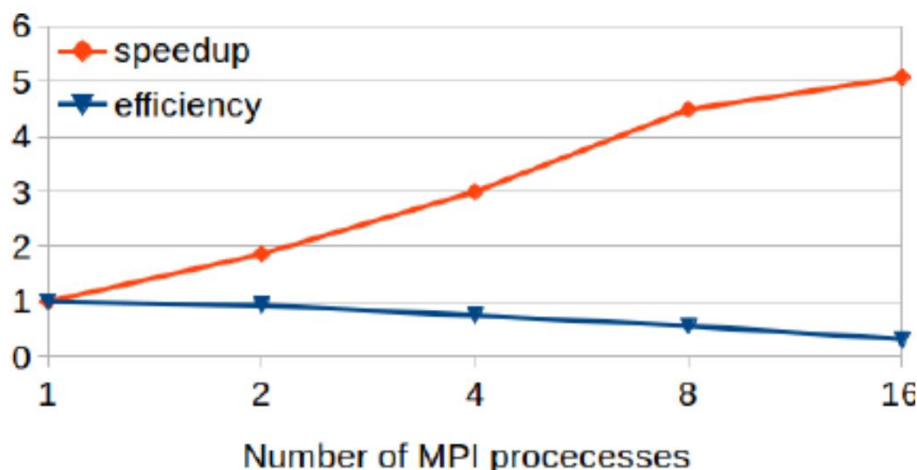
And the next iteration will be calculated as (i+1) multiplied by block size.

As the number of threads increase the amount of time for computing is decreased, it is due to the reason that the task is divided into different threads, and the less time for each thread is needed.

Our minimum time was 0.184 seconds with 16 threads, and maximum was 0.728 with 1 thread.

The parameters for image were 640 480 1.

The speed-up increases as the number of threads increases, however the efficiency decreases. Speed-up defines the degree of parallelism.



As we can see from the graph speed-up increases, while efficiency decreases.