Report on task 4

**Name: Alua Onayeva**

**Group: BD-2008**

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

**Main part:**

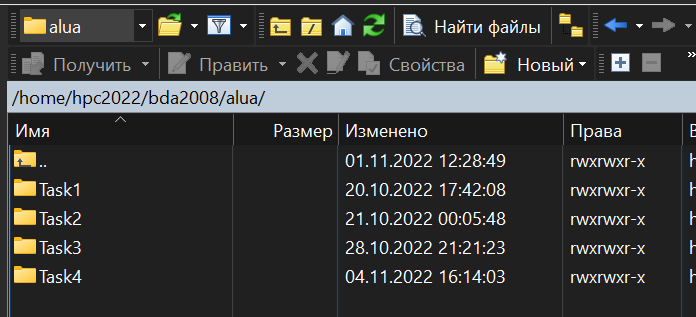
Step 0:

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 4

In your personal directory:

* Create directory “Task 4”
* Copy sequential program to this new directory
* Rename the file to raytracing\_mpi.cpp



Step 2: Implement parallel program with MPI

Use MPI to parallelize the sequential ray tracing program (edit raytracing\_mpi.cpp). In your parallel program:

1. Image to compute should be partitioned into blocks
2. Image blocks should be distributed among processes, processes should compute their block(s) in parallel
3. The main process (for example, rank 0) should receive computed blocks from all processes and create the final image file

You can implement any way of partitioning the image onto blocks and distributing blocks (a single block for each process or multiple blocks for each process, the same size for all blocks or different block sizes, static distribution of blocks or dynamic distribution, etc).

*Hint*: you can use [this program template](https://github.com/georgy-schukin/hpc-course/blob/master/task_templates/task4/raytracing_mpi.cpp) as a starting point.

*Hint*: study [this program example](https://github.com/georgy-schukin/hpc-course/blob/master/examples/mpi/array/array.cpp) about parallelizing computation of an array with MPI.

To compile MPI program:

mpicxx -O3 -o raytacing\_mpi raytracing\_mpi.cpp -lminirt

To run MPI program with N processes:

mpiexec -np *N* ./raytracing\_mpi <args>

Step 3: Study performance of your parallel program

1. Use MPI\_Wtime() to measure the execution time for the parallelized main loop on each process (the time to compute all image block(s) for a process), then compute the final execution time for the program (max time among all processes):

**double start = MPI\_Wtime();**

for(int x = …)

for(int y = …) {

    const auto color = viewPlane.computePixel(

scene, x, y, numOfSamples);

…

}

**double end = MPI\_Wtime();**

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

// Choose the maximal execution\_time among all the processes

//(use MPI\_Reduce for that)

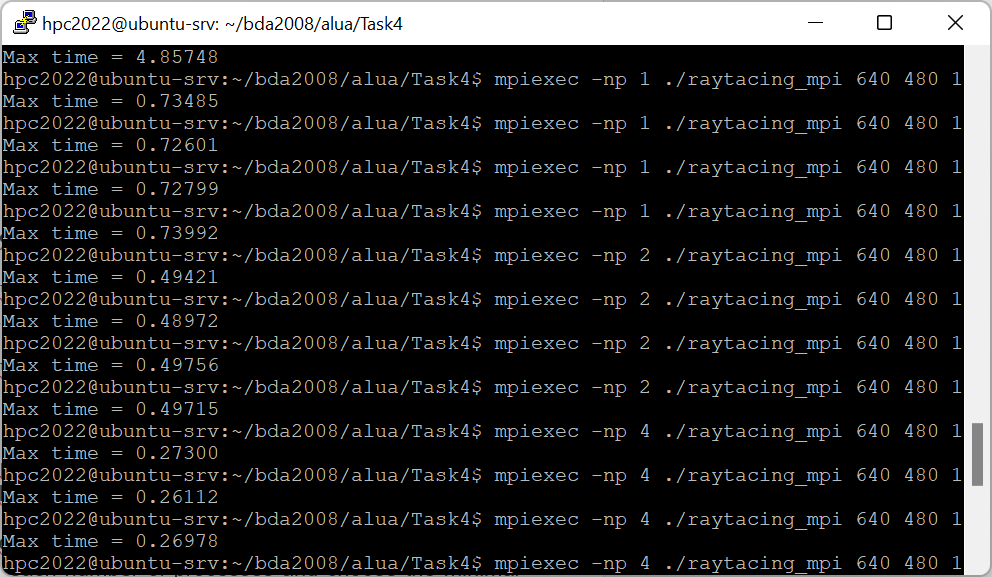
// Output the maximal execution\_time

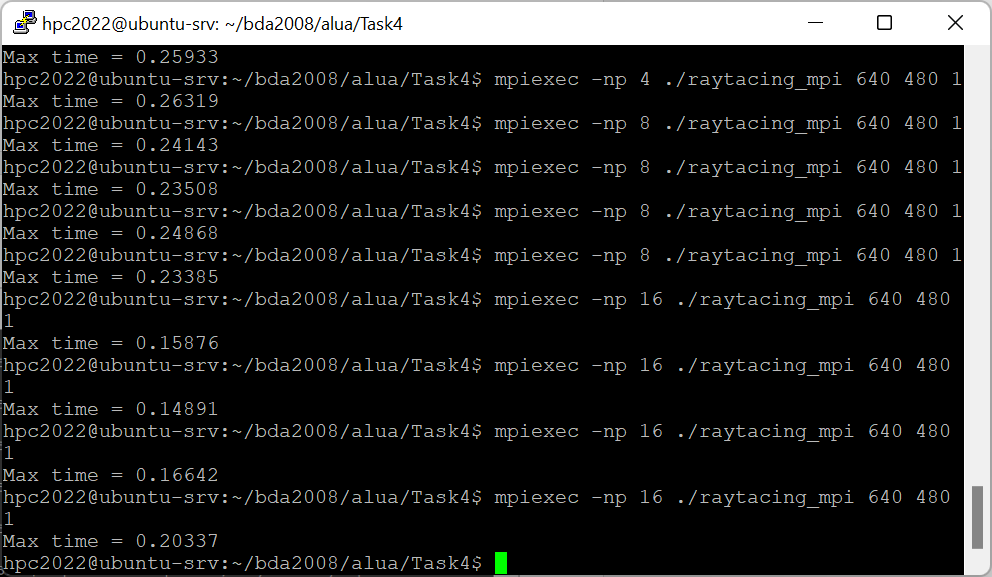
if(rank == 0) {

std::cout << “time = “ << max\_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 process, is more than several seconds.
2. Measure the execution time for the parallel program on 1, 2, 4, 8, 16 processes. For accuracy you can do several runs (>5) on each number of processes and choose the minimal time among runs for this number of processes.
3. Build plots/tables for:
   1. The execution time (to demonstrate how it depends on the number of processes)
   2. Speedup: Speedup(N) = Time(1) / Time(N), N - number of processes
   3. Efficiency: Efficiency(N) = Speedup(N) / N





|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Processes number | 1 | 2 | 4 | 8 | 16 |
| Execution time | 0.72601 | 0.48972 | 0.25933 | 0.23385 | 0.14891 |
| Speed-up | 1 | 1,4825 | 2,7995 | 3.1046 | 4,8755 |
| Efficiency | 1 | 0.74125 | 0.6998 | 0.3881 | 0.3047 |

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

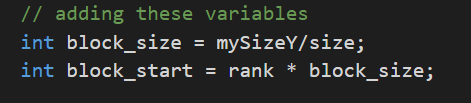
Upload your source code files in Task 4 directory to your Github repository, include a link to it in the report.

<https://github.com/loopiiu/hpc_task4.git>

Step 5: Conclusion in a free form

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

Briefly speaking, we have added these variables. Blocks are split on equal parts.



As we divide our array, indexes are being changed. To find our initial index we use i+block\_start

Изображение выглядит как текст

Автоматически созданное описание

The next condition computes maximum time of condition. MPI\_Reduce function takes an an input a lot of values, and as output gives one value, in our case maximum time. MPI\_DOUBLE type of the variable, here max\_time is a double type.

MPI\_COMM\_WORLD is a global communicator.

Изображение выглядит как текст, экран, снимок экрана

Автоматически созданное описание

Repeat initializing one more time.

Изображение выглядит как текст

Автоматически созданное описание

The same algorithm as before. srcSizeY is analogy of block\_size.

Изображение выглядит как текст

Автоматически созданное описание

In parallel programming speed up limited by many factors.

Speedup is generally limited by the speed of the slowest node or processor. Thus, we need to make sure that each node performs the same amount of work. or the system is load balanced.

So, in MPI performance is limited by the communcation network between the nodes.

The efficiency for 16 processes in my case is only 30%. I guess we need to use dynamic scheduling, instead of static one. As we divide program on processes of the same size.