

Name: Amir Yakubov
Group: BDA-2008
E-mail: 201463@astanait.edu.kz

Main part:

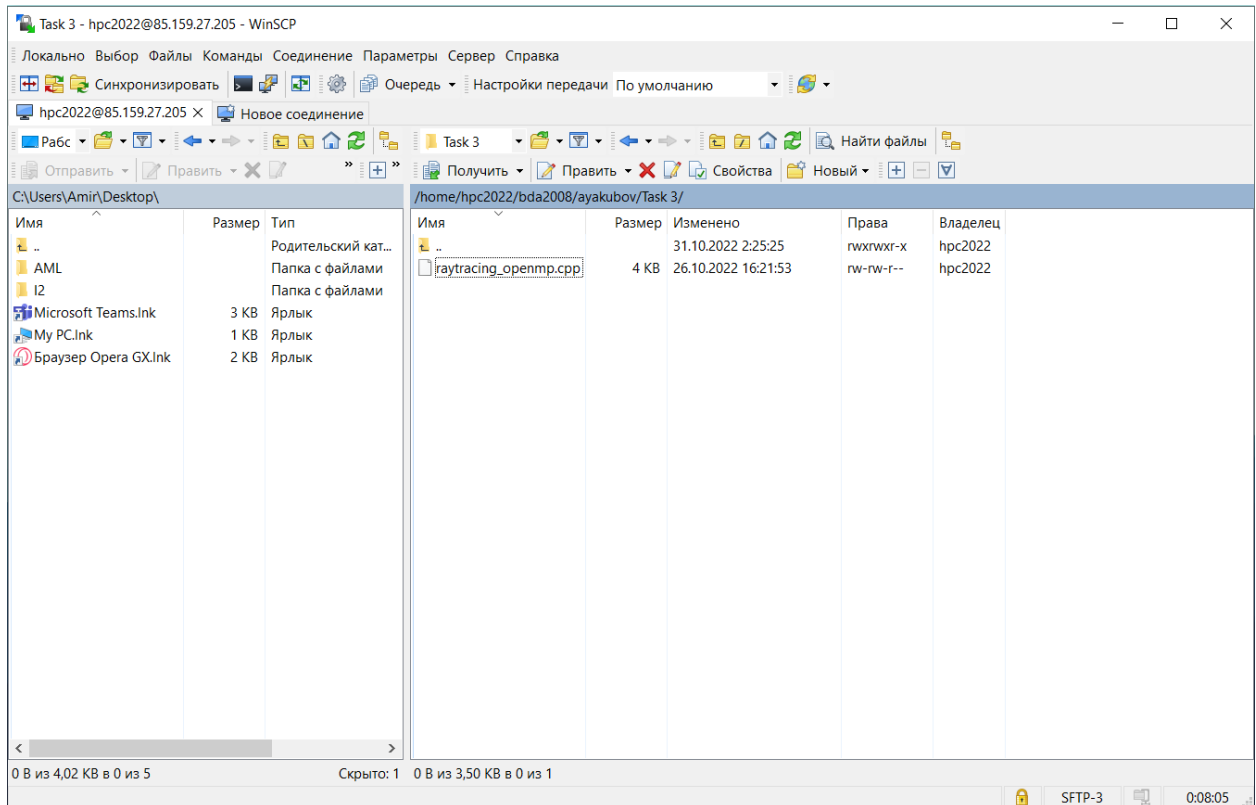
Step 0:

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



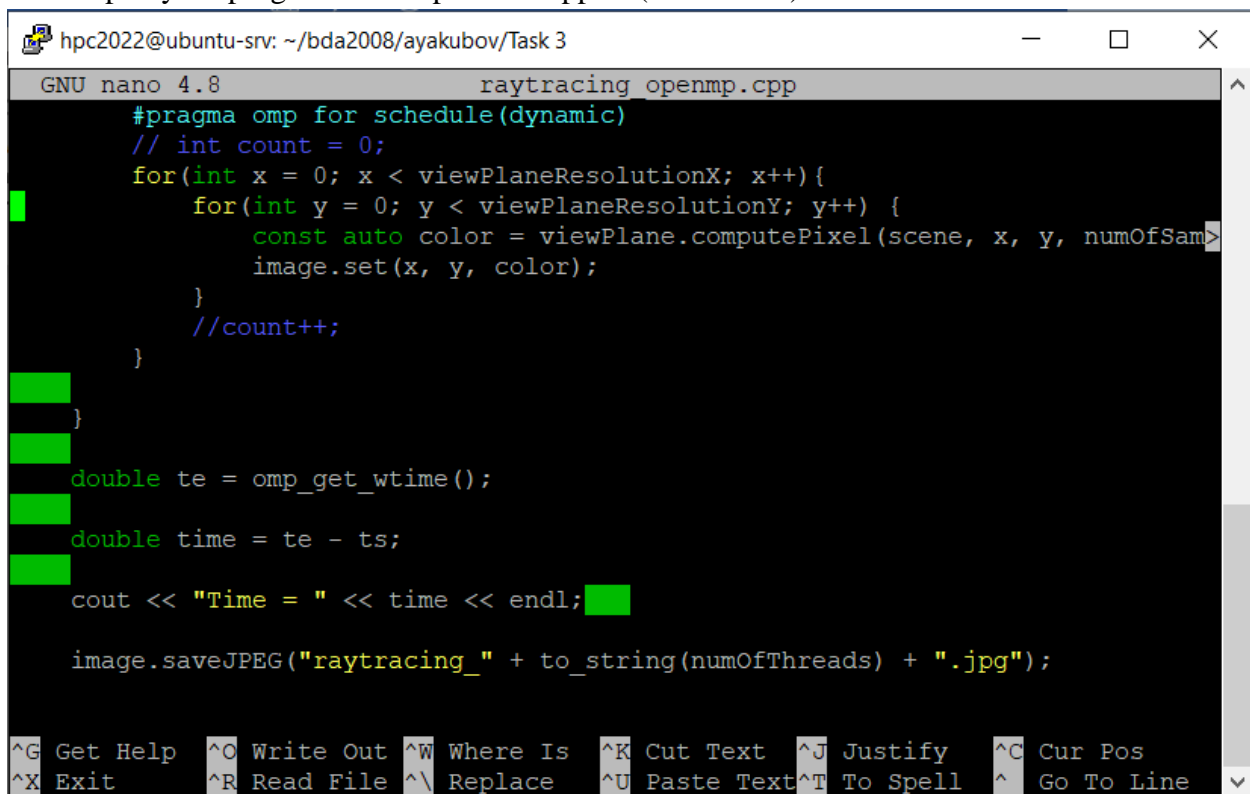
Step 2: Implement parallel program with OpenMP

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](#) as a starting point.

Hint: study [this program example](#) about parallelizing 'for' loop with OpenMP.

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



```
hpc2022@ubuntu-srv: ~/bda2008/ayakubov/Task 3
GNU nano 4.8 raytracing_openmp.cpp
#pragma omp for schedule(dynamic)
// int count = 0;
for(int x = 0; x < viewPlaneResolutionX; x++){
    for(int y = 0; y < viewPlaneResolutionY; y++) {
        const auto color = viewPlane.computePixel(scene, x, y, numOfSam
        image.set(x, y, color);
    }
    //count++;
}

double te = omp_get_wtime();
double time = te - ts;

cout << "Time = " << time << endl;

image.saveJPEG("raytracing_" + to_string(numOfThreads) + ".jpg");

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

`g++ -O3 -fopenmp -o raytracing_openmp raytracing_openmp.cpp -lminirt`

```
hpc2022@ubuntu-srv: ~/bda2008/ayakubov/Task 3
just raised the bar for easy, resilient and secure K8s cluster deployment.

https://ubuntu.com/engage/secure-kubernetes-at-the-edge

85 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

New release '22.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

*** System restart required ***
Last login: Sun Oct 30 21:04:35 2022 from 85.117.125.151
hpc2022@ubuntu-srv:~$ ls
aibekkyzy          bda2001  bda2004  bda2007  common      temp
AldiyarKenzhebayev bda2002  bda2005  bda2008  mini-rt     Zamanbek
AltynbekAbdiramanov bda2003  bda2006  build    mini-rt-build
hpc2022@ubuntu-srv:~$ cd bda2008
hpc2022@ubuntu-srv:~/bda2008$ cd ayakubov
hpc2022@ubuntu-srv:~/bda2008/ayakubov$ cd Task\ 3
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ nano raytracing_openmp.cpp
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ g++ -O3 -fopenmp -o raytracing_ope
nmp raytracing_openmp.cpp -lminiirt
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$
```

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;
```

2. 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.
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.

Static schedule

```

hpc2022@ubuntu-srv: ~/bda2008/ayakubov/Task 3
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ g++ -O3 -fopenmp -o raytracing_openmp raytracing_openmp.cpp -lminirt
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 1 640 480 10
Time = 7.18727
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 1 640 480 10
Time = 7.19286
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=2 && ./raytracing_openmp 2 640 480 10
Time = 3.70045
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=2 && ./raytracing_openmp 4 640 480 10
Time = 1.88174
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=2 && ./raytracing_openmp 8 640 480 10
Time = 0.960659
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=2 && ./raytracing_openmp 16 640 480 10
Time = 0.583808
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$

```

Static, 1 schedule

```
hpc2022@ubuntu-srv: ~/bda2008/ayakubov/Task 3
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 1 640 480 10
Time = 7.2152
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 2 640 480 10
Time = 3.69257
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 4 640 480 10
-bash: ./raytracing_openmp: No such file or directory
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 4 640 480 10
Time = 1.8865
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 8 640 480 10
Time = 0.973559
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 16 640 480 10
Time = 0.630141
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$
```

Dynamic schedule

```
hpc2022@ubuntu-srv: ~/bda2008/ayakubov/Task 3
hpc2022@ubuntu-srv:~/bda2008$ cd ayakubov
hpc2022@ubuntu-srv:~/bda2008/ayakubov$ cd Task\ 3
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ ls
raytracing_16.jpg raytracing_2.jpg raytracing_8.jpg raytracing_openmp.cpp
raytracing_1.jpg raytracing_4.jpg raytracing_openmp
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ nano raytracing_openmp.cpp
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ g++ -O3 -fopenmp -o raytracing_openmp raytracing_openmp.cpp -lminirt
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 1 640 480 10
Time = 7.19188
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 2 640 480 10
Time = 3.71028
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 4 640 480 10
Time = 1.88433
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 8 640 480 10
Time = 0.953725
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$ export OMP_NUM_THREADS=1 && ./raytracing_openmp 16 640 480 10
Time = 0.591264
hpc2022@ubuntu-srv:~/bda2008/ayakubov/Task 3$
```

5. 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: $\text{Efficiency}(N) = \text{Speedup}(N) / N$

Static schedule

Number of threads	Execution time	Speedup(N)	Efficiency
1	7.20417	1	1
2	3.70045	1.94683	0.973415
4	1.88174	3.82846	0.956215
8	0.960659	7.499195	0.937399
16	0.583808	12.339964	0.771247

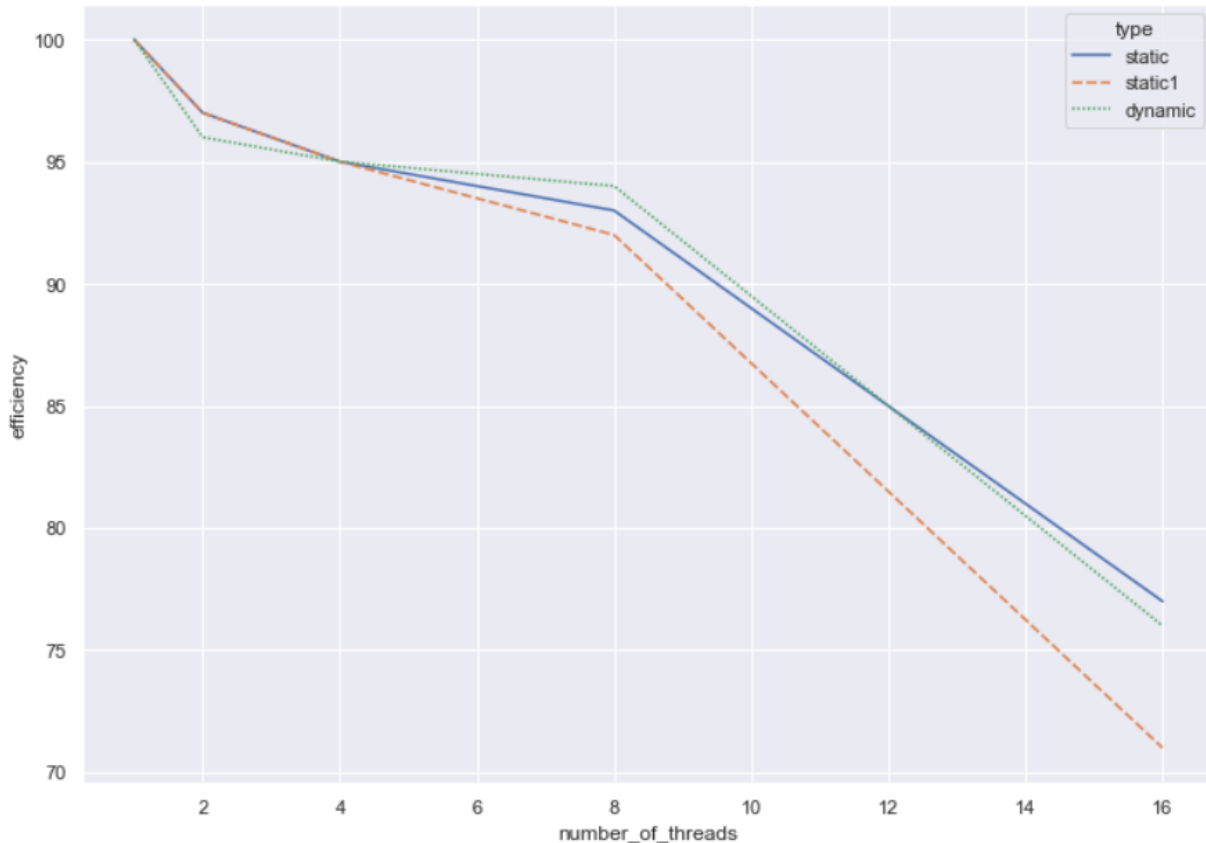
Static, 1 schedule

Number of threads	Execution time	Speedup(N)	Efficiency
1	7.2152	1	1
2	3.69257	1.95397	0.97397
4	1.8865	3.82464	0.95616
8	0.973559	7.41115	0.92639
16	0.630141	11.44013	0.71500

Dynamic schedule

Number of threads	Execution time	Speedup(N)	Efficiency
1	7.19188	1	1
2	3.71028	1.93836	0.96918
4	1.88433	3.81667	0.95416
8	0.953725	7.54074	0.94259
16	0.591264	12.16393	0.76024

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.



Efficiency in percents.

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/Am1rrr/hpc/tree/main/Task%203>

Step 5: Conclusion in a free form

Dynamic scheduling is acceptable when varied computing expenses are needed at various iterations. This suggests that there isn't a good enough balance between iterations and static methods. However, because it distributes the iterations dynamically while the program is running, the dynamic scheduling type has more overhead than the static scheduling type. The sole reason why dynamic scheduling is not much more efficient than static scheduling is that the time required to create an image is too brief to serve as an accurate comparison. It was probably worthwhile to do this in order to increase the picture's resolution or the sample count.