Advanced Computing – 2020/2021

MsC in Informatics – ESTiG/IPB

Practical Work: **Raytracer Acceleration**

Goal: To accelerate a simple raytracer, using the PThreads and/or MPI programming models.

Details:
"Ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a high degree of visual realism", though with considerable computational cost (see [1] for a broad overview the raytracing concept).

"Ray Tracing in One Weekend" (RTOW) [2] teaches the basics of coding a simple raytracer. It explains the main underlying concepts (you should read it at least once) and provides related source code in C++. There are, however, implementations in other languages, including in C. The base C implementation that will be the starting point for this practical work is available at https://cloud.ipb.pt/f/50cc1817d74040feb32b/?dl=1 (do not use any other version you may find).

To compile the C implementation given, a recent version of `cmake` is needed. The frontend of the ROCKS cluster used in the classes has such version in `/share/apps/cmake-3.19.3` . To use it, just add `source /share/apps/env_cmake-3.19.3` to the end of your `~/.bashrc` profile, then logout and login (you need to do this only once; afterwards, the new `cmake` version will always be used by default). Grab the RTOW archive from the link supplied above and extract it on your home folder. To compile it, execute:

```
        cd raytracing-weekend-46af5654eddc93b98be6176ae16821282c15ef23
        mkdir build
        cd build
        cmake ..
        make
```

You may then experiment running the raytracer with different options:
- see the usage

```
        ./ray_tracing_one_week --help
```

- render a sample

```
        ./ray_tracing_one_week --scene random_spheres random_spheres.ppm
```

To visualize the sample rendered, you may double click on the PPM image or execute:

```
        xdg-openrandom_spheres.ppm
```

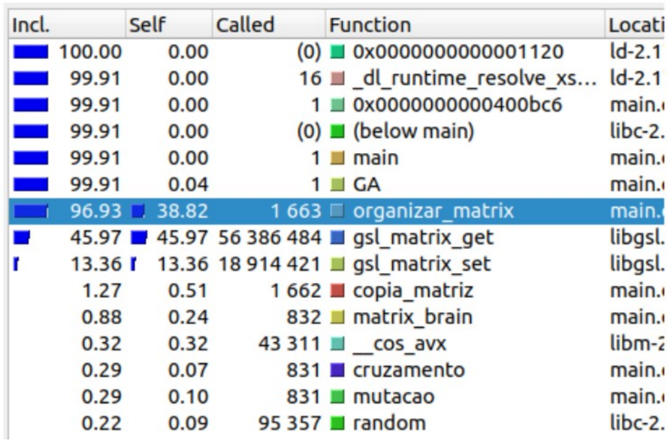
There are 9 scenes to choose from and each takes a different amount of time to render. This time is determined by the number of rays to cast for each pixel, which is 1000 by default. You may change this value using the `-s` parameter (`./ray_tracing_one_week -s 10 …`).

You will find that the rendering process may take a lot of time (to measure the rendering time just execute `time ./ray_tracing_one_week ...`). This is because in the supplied C implementation everything is done serially (there is none parallelism employed). Therefore, the goal of this work is to accelerate the supplied raytracer, by exploiting the parallel programming techniques learned in the classes, while keeping the correctness of the images generated. At minimum, you should develop a PThreads version or a MPI version. But you may find beneficial, performance-wise, to combine both approaches (hybrid programming).

Once acceleration is the primary goal, the solutions of the different groups will be ranked in terms of the performance achieved and some of the grading points will be assigned accordingly. The performance metric will be the overall time to render all 9 scenes, using at most 8 working nodes of the ROCKS cluster (the frontend node is not to be used) and a maximum of 32 CPU-cores per node.

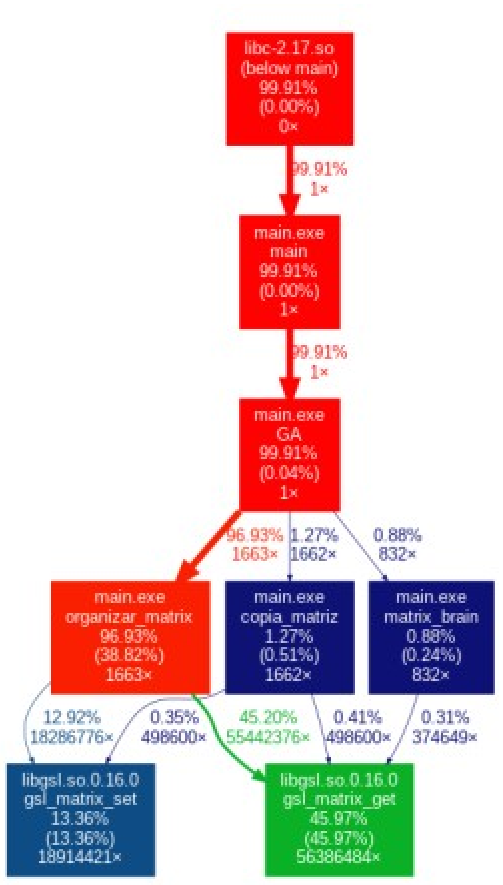
Start by studying the source-code. This visual inspection should help you to detect blocks of code that can be easily parallelized. Beware, however, that not always pays off to parallelize. Thus, before starting to change the original code, you should profile the code, in order to identify the main *hot-spots*. These are the code blocks that are more CPU-intensive and thus should be parallelized first. To profile your code, use the Callgrind tool of the Valgrind framework [3] to generate profiling data, and use kcachegrind [4] to visualize that data. Use also gprof2dot [5] to generate a call graph. The output of these tools should be in your report. The images bellow are output examples. Note: valgrind and kcachegrind are available in the cluster frontend; for gprof2dot, install it in your home.

Example output of kcachegrind:



Incl.	Self	Called	Function	Locati
100.00	0.00	(0)	🟩 0x0000000000001120	ld-2.1
99.91	0.00	16	🟥 _dl_runtime_resolve_xs…	ld-2.1
99.91	0.00	1	🟩 0x0000000000400bc6	main.
99.91	0.00	(0)	🟩 (below main)	libc-2.
99.91	0.00	1	🟨 main	main.
99.91	0.04	1	🟩 GA	main.
96.93	38.82	1 663	⬜ organizar_matrix	main.
45.97	45.97	56 386 484	🟦 gsl_matrix_get	libgsl.
13.36	13.36	18 914 421	🟩 gsl_matrix_set	libgsl.
1.27	0.51	1 662	🟥 copia_matriz	main.
0.88	0.24	832	🟨 matrix_brain	main.
0.32	0.32	43 311	🟩 __cos_avx	libm-2
0.29	0.07	831	🟦 cruzamento	main.
0.29	0.10	831	🟩 mutacao	main.
0.22	0.09	95 357	🟩 random	libc-2.

Example output of gprof2dot:



After using these tools (and possibly instrumenting your code to measure the time taken by the most relevant blocks) you should have a clear view on what code should be parallelized and how much time does that code consumes in the sequential version. Apply Amdahl's Law to make a projection of the ideal speedups and write that in the report.

As you start parallelizing your code, keep a record of your progress. In the report, the progresses made should be clear: what were the sequential running times, what changes were made, and what was the Speedup and Efficiency achieved.

Deadline:
Reports (in PDF) should be sent to rufino@ipb.pt until February 14th 2021.

References:

[1] https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
[2] https://raytracing.github.io/books/RayTracingInOneWeekend.html
[3] https://www.valgrind.org/docs/manual/cl-manual.html
[4] https://kcachegrind.github.io/html/Home.html
[5] https://pypi.org/project/gprof2dot/