A parallel ray tracer

April 21, 2022

Introduction The project has to be done by group of two (preferred) or alone (please ask). A short report (one page) with the requested information has to be provided. The objective of the project is to get a practical knowledge of parallel programming on a non-toy example.

The goal is to modify a simple ray tracer program to make use of parallelism, in particular in multicore computers. The program is provided, with the source, on the eCampus space of the course (I am grateful to Dmitry Sokolov for providing a first basis for this software). In principle, you do not have to bother with geometrical aspects. All language and GUI library aspects (e.g., interacting with the keyboard, loading files, displaying files) are already demonstrated in the provided sequential program. It is not asked to improve the running times by modifying the ray tracing algorithm.[1] The work essentially focuses on parallelism aspects only.

The provided code The provided code is in C++. It is not particularly well written, well organized, or conceived to be parallelism-friendly. It is part of the project to get along with non-optimal code. To compile the software on Linux, you need `gcc` and `SFML`. It works on the ms8xx network, and it is fairly possible to run remotely, even with the graphical interface (you will have to configure an X11 server on your machine, whatever it is, and use X11 forwarding with ssh). It is also possible to run using Windows Subsystem for Linux. If you are able to compile and run directly on Windows or MacOS, we are grateful if you can provide us with a short tutorial.

Currently, the program supports three options:

- `-gui` opens a windows where the rendering is showed;
- `-animate` animates the rendering;
- `-full` is short for both previous options.

The user interface allows to change the viewing angle using the keyboard arrows, and "q" exits the software. Without gui, a run just renders the scene and saves it into a jpeg file.

Your mission (should you choose to accept it) The sequential program is usable, and interaction is working. However, interaction with the user is clumsy (e.g. press on the left key for some long time, and see what happens), and no usage is made of the multicore architecture of modern computers, resulting in a low frame count per second. Furthermore, there is no control of the frame rate, the speed of the animation being a function of the speed of the computer.

The objectives of the project are to learn how to

1. use the available cores on the computer in order to increase the amount of computation per unit of time, i.e., to increase the number of frames per second (fps);
2. organize code to decouple computation and interaction aspects in the program, for a better user experience;

[1]Although there are numerous opportunities to do so (e.g., avoid repeated computation of the same values for pixels which do not change, having a better selection of objects that are in the way of the ray, improving computations,...) please do not waste time on improving the ray tracing computation, since we will not evaluate that in any way.

3. understand that the structure of the program has to be conceived with parallelism in mind, e.g., functions should be thread safe;
4. use a cluster of computers, again to increase the number of frames per second.

According to the objectives above, the project has several phases:

1. Use openMP to improve the frame count per second. This should be a one line change in the code, so that the computation of one image is done in parallel. It might take time for you to locate where to operate the change, but if it needs more than one line, you are doing it wrong. This simple phase serves as an entry point into the code. The previous hands-on exercise on OpenMP should have prepared you for this easy task.

   Rather than modifying the code, introduce a new rendering function that works in parallel for the computation of one frame. It will be useful for the subsequent phases.

2. In the first phase, several threads compute a same image. This allows to make use of parallelism in an easy way, but this has limitations. In particular, it is not extensible for use on a cluster of machines rather than a multicore machine. Now, we will use different threads to compute different images. Use one thread for displaying images and dealing with user interaction. Another thread will be a manager for the computation of images, and a certain amount of threads ($n - 1$, where n is the number of real cores in your machine) for the proper computation. Use a thread pool, that is, do not create/destroy a thread each time an image is computed. Remember the model of the producer-consumer on a circular list to organize the exchange of information, and control the computation of future frames as well as the frame rate. The logo should rotate at a frequency of one rotation per minute, whatever the frame rate and the power of the computer.

3. In the version of the second phase, make the red sphere oscillate vertically (between -1 and 0), and the mirror sphere oscillate in size (between 3 and 4), with a period respectively of 5 and 3 seconds.

4. The scenery as it is by default is reasonably quickly computed. It might happen that the work per image is much larger (uncomment for instance the `RENDER_DUCK` macro definition at the beginning of `tinyraytracer.cc`, compile, test, and check the fps). In that case, it might be useful to use a cluster of machines to compute images, because it is becoming advantageous to use the network to transmit images. Use MPI to also use distant computers to help computing frames. If you use MPI together with threads, remember to initialize MPI with the `Init_thread` version of the initialization function.

5. Impress us (but remember that only the parallelism aspects are evaluated).

Deliverables. In the short report (one page), explain what you did for phases 2, 3, and 4 (no need to explain phase 1). Report the number of real cores on your experimentation machine, and the number of threads you used for the computation. Provide fps counts for the (given) sequential version, and your version at the several phases. If you are proud of some feature of your program, do not forget to mention it in the report!

A zip file providing a snapshot of your program (with source) for each phase has to be uploaded to eCampus.

Evaluation. You can reach 13/20 with only phases 1 and 2, and 15/20 with only phases 1 to 3, if those phases are nicely done.

Organization The last sessions of the course are dedicated to help you tackle all tasks in this project. Please do not leave all the work for the last days, because it will increase the total amount of time you will have to invest in this project.

Cores, hyperthreading, and all that. There are system dependent ways (i.e., it differs whether you are using Windows, Linux, or another system) to allocate a thread to a particular logical cpu. This means you can ensure that the computation of frames is done on different cores. We do not ask you to use system dependent features, because it will make checking your code difficult. Leave the scheduling of the threads to the kernel, and use a macro to specify the number of cores the program should use.

Plagiarism. Cooperation (even between groups) is allowed, but all cooperation must be clearly referred to in the report, and we will have **no tolerance at all for plagiarism** (writing together, reusing/sharing code or text). Code available on the web that serves as a source of inspiration must also be properly referred in the report.

Participation. **Each student has to participate** in the project, and do her/his fair share of work. Letting others in the group do all work will be considered as plagiarism.

Important final remark. This project is not meant to be a huge amount of work, or to let you find by yourselves, e.g., obscure options of MPI. Ask for help, come to the hands-on session on Fridays, use the Forum, request a Q/A zoom session, if needed.