UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 2: Ray Tracing with OpenMP

Year 2023–2024

Contents

1 Introduction

Ray Tracing is a technique widely used to generate synthetic computer images. In this Lab Unit we start from a sequential program (`ray.c`) which implements a very basic version of a ray tracing algorithm. The program is a translation to C language (with slight modifications) of the basic ray tracing program (in C++ language) offered by `scratchapixel`[1].

The objective of the Lab Unit is to parallelize this program using OpenMP, exploring different alternatives and comparing the performances obtained.

2 Ray tracing technique

The Ray Tracing technique for image generation consists of tracing rays from the observer's point of view in the suitable directions to obtain the colors of all points in the image. For each ray, you must check which objects in the scene it intersects and get the closest intersection. The color of that closest intersection will give the color of the corresponding image point. To calculate the color of the intersection with an object, it will be necessary to draw new rays if the object is partially transparent or reflective. For this reason, this part of the algorithm is usually implemented using a recursive function that can call itself multiple times.

[1] scratchapixel (`http://www.scratchapixel.com/`), a web resource that allows you to learn both 2D and 3D computer graphics techniques starting from scratch.

The algorithm 1 presents a (very) abridged version of the ray tracing algorithm written in pseudo-code. This algorithm partially corresponds to the `render` function of the provided program.

Algorithm 1: Ray tracing

```
1 for y = 0, ···, height − 1 do
2     for x = 0, ···, width − 1 do
3         Calculate (xx,yy,zz) 3D coordinates of the point (x,y) of the image plane to be obtained.
4         Draw a ray from the camera in the direction of the point (xx,yy,zz) and look for the closest
            object hit by the ray.
5         Calculate the color of this object at the hitpoint (which may involve tracing more rays
            recursively if the object is reflective or transparent).
6         Paint the point (x,y) of the image with the calculated color.
7     end for
8 end for
```

In this function, it is important to understand the data type of the variables involved. In particular it is important to understand that:

- `Vec3` is a structure that contains three real numbers (`double`) in the attributes `x`, `y`, `z`. This structure is used both to represent a point or vector in space and to represent the color of a pixel (values of the three RGB channels).
- `image` is a dynamic vector containing `width`× `height` elements of type `Vec3` that represent the color of each pixel in the image.
- `pixel` is a pointer to an element of type `Vec3` (a pixel of the image).

Note that for each pixel of the image an invocation to the `trace` function is made, but this invocation can cause other invocations recursively. The `trace` function returns the number of invocations that had to be made for the pixel, which indicates the complexity of the pixel calculation.

The function `render` obtains, for each line of the image, its complexity, expressed as the average number of invocations to `trace` per pixel (variable `ncalls_line`) and, from this information, calculates the maximum/minimum complexity among the lines (`max_ncalls_line`, `min_ncalls_line`). Additionally, you get a histogram (vector `histo`) that indicates, for each complexity interval, the number of lines that lie in the interval.

2.1 Using the program

The program generates an image using the ray tracing technique for a scene with several randomly placed spheres. The number of spheres in the scene can be given as an argument. If the argument is omitted, 6 spheres will be considered. For example, for a scene with 40 spheres:

```
$ ./ray 40
Computing...

Complexity histogram:
 Calls/pixel    |  Number of lines
 ---------------------------------------------------------------------
 [ 1.0 --  2.5[ |  196 **************************************************
 [ 2.5 --  4.0[ |   62 ****************
 [ 4.0 --  5.5[ |  108 ****************************
 [ 5.5 --  7.0[ |  154 ****************************************
 [ 7.0 --  8.5[ |  148 ***************************************
 [ 8.5 -- 10.0[ |  100 **************************
 [10.0 -- +inf[ |    0
```

```
Least complex line had  1.000 calls to trace per pixel.
Most  complex line had  9.822 calls to trace per pixel.
```

The resulting image is saved in the current directory in a file with the name `untitled.ppm`. In Figure 1 you can see an image generated by the program. The program also shows information about the complexity of the lines: maximum/minimum complexity of the lines and histogram.

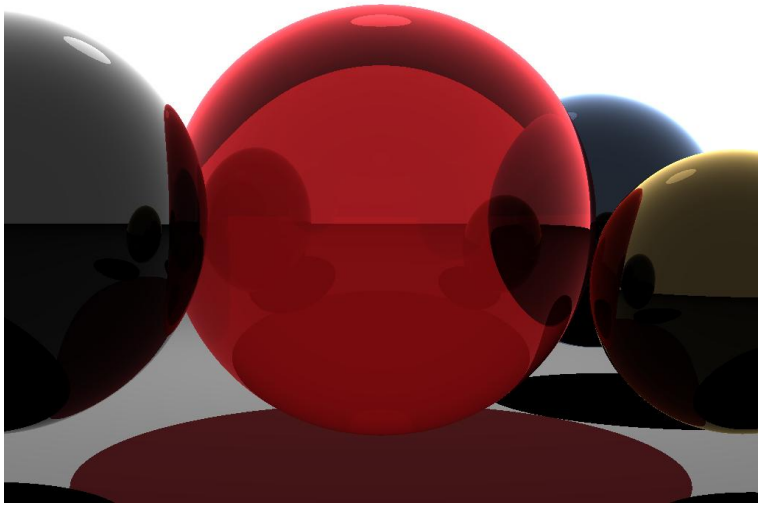

Figure 1: Image generated by the program.

Compile and run the program and look at the generated image. Try a different number of spheres.

3 Development of parallel versions using OpenMP

Before considering parallelization, start by modifying the original code to display the time spent to generate the image (image generation only, not including the part that writes to the screen and to the file). Although it is a sequential code, use the OpenMP function to measure the time. Save this version as `ray0.c`. It will be used later to compare the time of the sequential program with that of the parallel programs.

The following exercises ask you to develop two parallel versions of the program. Both versions must show, in addition to the execution time, the number of threads with which they are executed. Justify in the report the choices you made about the scope of variables (private, reduction), as well as the possible use of synchronization directives.

We will focus on the parallelization of the `render` function, which contains the loops that calculate all the points in the image. It is important to analyze the code of this function as well as the variables involved.

Exercise 1: Parallelize the inner loop of the `render` function (`x` loop) using OpenMP. This version should be called `rayp1.c`.

Check that, in the basic case (execution without arguments), the generated image is **exactly** the same as the one obtained with the sequential program (use the commands `cmp` or `diff` to compare files, as indicated in Lab Unit 1). Also check that the information shown on the screen about complexity (histogram and least/most complex lines) is the same as in the sequential program.

Exercise 2: Parallelize the outer loop of the `render` function (`y` loop) using OpenMP. This version will be `rayp2.c`.

In this version, special attention must be paid to the function `update_histogram`, which updates the histogram vector (`histo`) taking into account the number of calls necessary for the current line of the image.

Check again that the generated image and the complexity information are exactly the same as in the sequential case.

4 Performance study

The following exercises ask you to analyze the performance by measuring the execution time using the `kahan` cluster queuing system. A sufficient number of spheres must be used (for example 600) so that the computational load is large. Add to the report the number of spheres used.

Exercise 3: In this exercise you have to evaluate the influence of the iteration scheduling on the performance of the parallel versions.

Using the `kahan` cluster queuing system, obtain the execution times for the two parallel versions made using 32 threads and different scheduling. Specifically, use the following schedule choices:

- `static` with default chunk size.
- `static` with chunk size equal to 1.
- `dynamic` with default chunk size.

Analyze which is the best/worst schedule choice for each parallel version, justifying the possible cause. Connect it to the concept of **load balance** and, if you consider it appropriate, to the histogram produced by the program.

Exercise 4: Using the `kahan` cluster queuing system, obtain execution times for the two parallel versions, varying the number of threads and choosing for each version the schedule choice producing the best results in the previous exercise. To limit the number of runs, it is recommended to use powers of 2 for the number of threads (2, 4, 8...), reaching up to the number of threads that you consider appropriate.

Show tables and graphs for times, speed -ups and efficiencies, and use them to compare the performance of the two parallel versions. In view of the results, draw conclusions indicating which is the best parallel version or if they have similar behavior, and try to justify the results you obtain.

Exercise 5: Explain how you launched the executions in the cluster for the previous exercises, indicating how you establish the number of threads and the schedule choice and attaching some of the job files used.

Justify the maximum number of threads that you have considered in the previous exercise. Connect it to the hardware used.

5 Additional exercises

Exercise 6: Taking version `rayp2.c` as the base, add the necessary instructions so that:

- In addition to showing the number of calls to `trace` of the most complex line, the program indicates which line is the most complex. Ditto for the least complex line.
- Each thread indicates the number of calls to `trace` that the thread makes (including recursive calls) in total.

This version will be `racep3.c`. For example, if we run the program with 8 threads, we might get something like the following:

```
Computing...
Thread  4: 904056 calls to trace
Thread  0: 185243 calls to trace
Thread  1: 446134 calls to trace
Thread  5: 750639 calls to trace
Thread  7: 104451 calls to trace
Thread  2: 567047 calls to trace
Thread  6: 523485 calls to trace
Thread  3: 710522 calls to trace

...( Irrelevant lines omitted ) ...

Line  711 was the least complex, with  1.000 calls to trace per pixel.
Line  392 was the  most complex, with 11.489 calls to trace per pixel.
```

Justify in the report the decisions made to implement this version. Justify whether the number of calls made by each thread varies as the schedule varies.

6 Delivery instructions

The report and source code to be delivered in this laboratory exercise must be the **personal and original work** of the corresponding group (of a **maximum of two students**, both belonging to the **same laboratory group**). Therefore, copying any part of the report or code is strictly forbidden and corrective measures will be applied.

There are **two assignments** in PoliformaT for the delivery of this Lab Unit:

- In one of the assignments, you must upload a file **in PDF format** with the report describing the exercises and answering the questions. Other formats will not be accepted.
- In the other assignment, you must upload a single compressed file with the source code files of the different versions that you have developed, along with some of the job files used to launch the executions. **Do not include the executables resulting from the compilation or image files**. The file must be compressed in `.tgz` or `.zip` formats.

  Check that all files compile correctly and follow the naming specified in this document.

When completing the assignments, take into account the following recommendations:

- A descriptive report must be submitted of the codes used and the results obtained. Make sure the report is of a reasonable size (not a couple of pages, not several dozens).
- Exercise 6 represents 20% of the work grade (0.3 points). You can choose not to submit it, but in that case your maximum grade will be 1.2 points.
- Do not include the complete source code of the programs in the report. You can include, if you wish, the fragments of the code that you have modified.
- Take special care to prepare a good work report. It is not a mere presentation of results, but the report must have an adequate structure and contain the necessary analyses, explanations and conclusions. Read carefully the actions requested in each exercise.