Circle rendering

Arianna Carletti
E-mail address
arianna.carletti1@edu.unifi.it

Abstract

The focus of this paper is the implementation of a pipeline for rendering semi-transparent circles in relation to a z-coordinate. Since this problem has a considerable computational cost, both a sequential and a parallel version had been implemented. The problem is approached in three steps: first, mapping circles into the sections on which they lie, second, mapping circles into pixels, and finally, calculating the pixel value. This subdivision makes it possible to limit synchronization in the initial part, helping parallelism. Finally, a set of tests was conducted to analyze the speed increase caused by parallel execution as the parameters provided varied.

1. Introduction

Generally, rendering refers to the pipeline that provides the correct two-dimensional view, given a certain point of view, of a tridimensional scene. However, in this paper, we deal with the rendering of bidimensional semi-transparent shapes (circles in our case) that lie in three dimensions, and which therefore have a relative order that must be preserved and correctly represented in the generation of the rendered image.

We represent circles by the size of their radius, the position of their center, expressed in terms of abscissas and ordinates (x and y), a third coordinate (z), indicating circles' relative depth and their rgb values. This ordering expresses the layer on which a circle lies, and consequently, in the case of overlapping circles, allows to render them correctly, i.e., drawing the circle with the higher z coordinate in front of the one with the lower coordinate.

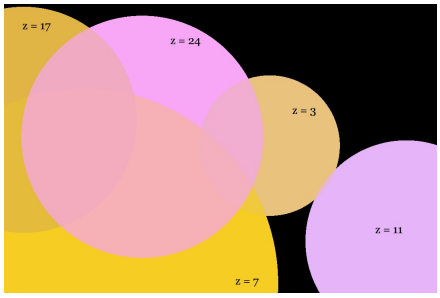

Figure 1. Rendering of 5 overlapping circles with different z coordinates.

Another crucial aspect to consider is that the center and radius of the circle are represented by real numbers, while the image on which they are drawn is obviously composed of pixels. This implies the need to determine whether a pixel belongs to a circle, in other words, if the center of the pixel is contained in the circle.

The generation of the rendered image takes place in three phases, as can be seen in Figure 2:

- *Mapping circles to sections*: the image is divided into sections in which circles are mapped. This step is intended to make the computation embarrassingly parallel in the successive phases, because of this, it is necessary to insert a circle in each section touched by it and not only in that its center lies.

- *Mapping of circles to pixels*: calculate pixels inside a circle and contained in a section. In this way to each pixel are associated all circles that lie on it and that will contribute to the final color of that pixel.

- *Calculation of pixel color level*: at this point the final color of a pixel is calculated by adding the contributions of all circles and weighing them for values that represent their transparency.

It is easy to notice that this process is computationally expensive as it depends on both the number of circles to render and the size of the image, so we decided to approach the problem with a parallel execution based on OpenMP.

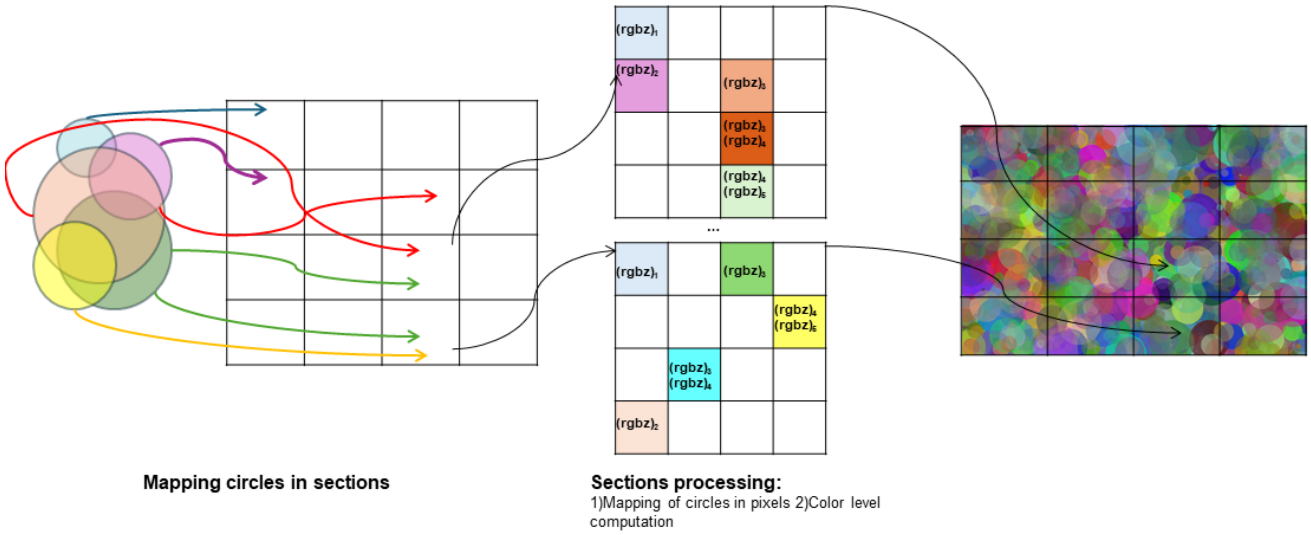

Figure 2. On the left we can see circles mapped in sections and then, each section is processed independently, building a matrix to store rgbz value for each pixel. At the end the rendered image is generated processing pixels independently.

2. Pipeline

2.1. Circle section mapping

The first step of the rendering pipeline consists in dividing the image into parts and assigning to each of them all and only the circles that lie on at least one pixel of that section, so it can then happen that a circle is mapped on several sections. This will enable the sections to be handled independently, leading to the problem being treated as embarrassingly parallel from now on.

2.2. Circle pixel mapping

In the next step, we then process sections, and for each of them, we consider the circles contained in them and calculate the pixels in the section covered by them. To check whether a pixel belongs to a circle, it must be verified that the center of the pixel has a distance from the center of the circle less than or equal to the radius of the circle; this would involve calculating this distance for all pixels contained in a square of dimensions equal to the diameter. However, the pixels inside the square inscribed in the circle can be considered as certainly belonging to the circle and actually calculate the distances for only the pixels in the frame between the square that inscribes the circle and the one that circumscribes it.

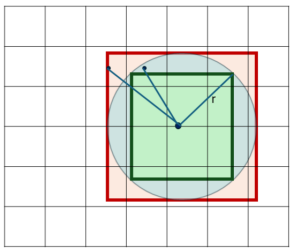

Figure 3. Pixels in the green area surly belongs to the circle, it's necessary to calculete distances only for pixels in the red area.

This step requires a matrix of dimension $h * w * num_c * 4$, where w and h are section width and height, num_c is the number of circle laying on that pixel and 4 states for RGBZ value of the circle. At the end of this step, the matrix contains all the information needed to compute the rendered image.

2.3. Pixel color computation

Then follows the color level pixel calculation; first of all, we compute the weights associated to each circle referring to the circle depth layer; we do this counting the number of circle on a pixel and then computing the multiplication factors following the alpha blending formula:

$$c_3 = \alpha \cdot c_1 + (1 - \alpha) \cdot c_2$$

where alpha states the transparency factor, c_3 is the resultant color level, c_1 is the color level of the circle on the upper layer and c_2 includes all the layers below. This formula is applied recursively to estimate the correct factor for each layer.

3. Parallel version

The process described above requires a significant computational cost. The circle-section mapping step depends on the number of circles, whereas the next two phases depend on number of sections, number of circle in a section and number of pixels. So, a parallel version had been implemented using OpenMP.

First of all, for each circle, are computed in parallel the sections they belong to. Circles are stored in an array of dimensions equals to the number of sections, each element of this array is a vector of circles. To improve parallelism, each thread has a local copy of this array and all copies are merged at the end of computations, using the pragma omp single directive. The pseudo-code for this first phase is in listing 1.

Algorithm 1: Parallel mapping of circles in sections

```
Initialize global array of circles organized in
  sections;
#pragma omp parallel;
Initialize local copy for this thread;
#pragma omp for
    for i ← 0 to NUM_CIRCLES − 1 do
        Calculate on which sections this circle is
          located;
        Add sections to local copy;
    end
end
#pragma omp single;
Merge local copies;
```

From now on, we can consider each section independent from the others, so we can parallelize section computations without needing of synchronizations.

So, for each section, we first of all initialize the matrix to store all the RGBZ value of the incident circles on a pixel, and then we process all the circles in that section, calculating the pixels inside the circle. Doing this, we fill the matrix adding RGBZ value with ordered insertion based on Z coordinates. Now we can finally calculate pixel color level processing the RGBZ matrix. For each pixel we count the number of circles laying on it, compute all the alpha value layer for transparency and at the end we multiply each circle color level contribution with the alpha transparency factor associated to his layer of depth. The pseudocode to summarize what had been said is in listing 2.

Algorithm 2: Parallel section processing

```
#pragma omp parallel for collapse(2)
    for h ← 0 to numSectionsHeight do
        for w ← 0 to numSectionsWidth do
            Compute real section width and height
              to handle edges of the image;
            Initialize RGBZ matrix;
            foreach circle ∈
              circlesInSections[SectionId] do
                Find pixels belonging to the circle;
                Convert local coordinates of the
                  section to general coordinates of
                  the image;
                Add circle to RGBZ matrix
                  according to Z coordinates;
            end
            for i ← 0 to
              current_section_height − 1 do
                for j ← 0 to
                  current_section_width − 1 do
                    Compute transparency factors;
                    for c ← 0 to circlesInPixels
                    do
                        Multiply all RGB value for
                          them layer transparency
                          factor and sum all
                          contributions.
                    end
                end
            end
        end
    end
end
```

4. Testing

Some tests are conducted to analyze the speedup curve when parameters vary. In all tests, circles are generated randomly beforehand and the same set of circles is used for both sequential and parallel version. All rendered images are of dimension 1080x720. The reported times are averaged over 20 tests, and for the parallel version the tests were performed with a number of threads ranging from 1 to 32 (double those available on the device on which they were run). The parameters varied in the tests are the radius dimension, the number of sections and the number of circles.

In the following plot are reported the computing time for sequential and parallel algorithms.

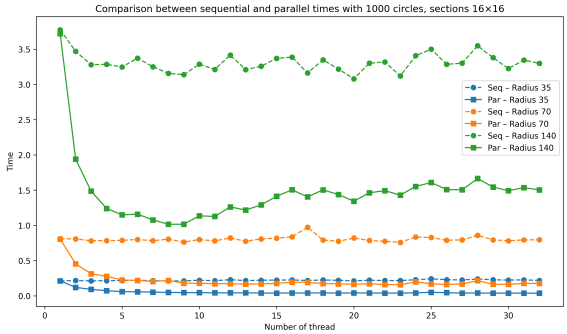

Figure 4. Comparison of parallel and sequential time (in seconds) to render an image with 1000 circles and 16x16 sections.

Figure 4 demonstrates how radius dimensions affects parallelization; in fact, with big circles covering several sections, we are mapping several times the same circles.

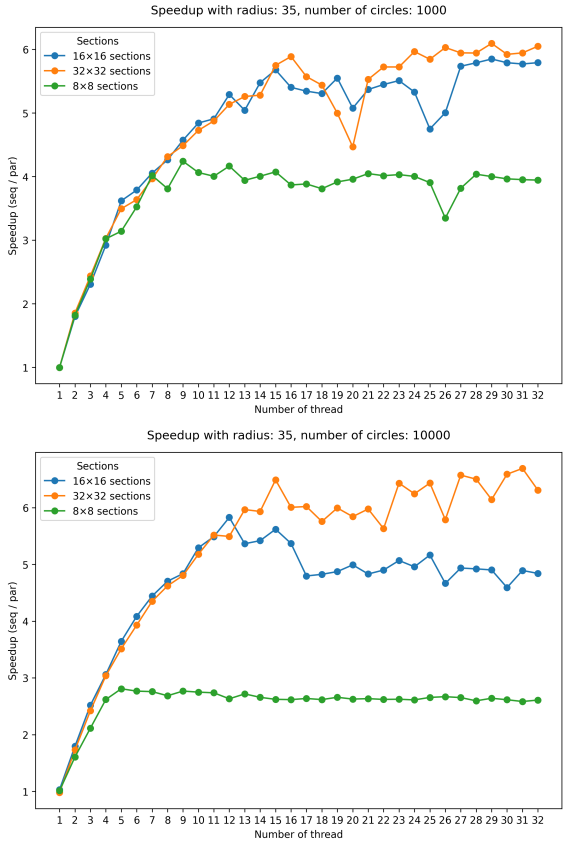

Figure 5. Speedup with 1000 and 10000 circles of radius not bigger than 35 pixels as the number of sections varies.

In plots from Figure 5, 6, 7 and 8 we can see the speedup calculated varying radius, number of sections and number of circles; sections should be interpreted as being arranged to form a rectangle, therefore with 16x16 sections are meant 256 sections that cover the image.

Figure 6. Speedup with 1000 circles of radius not bigger than 70 pixels as the number of sections varies.

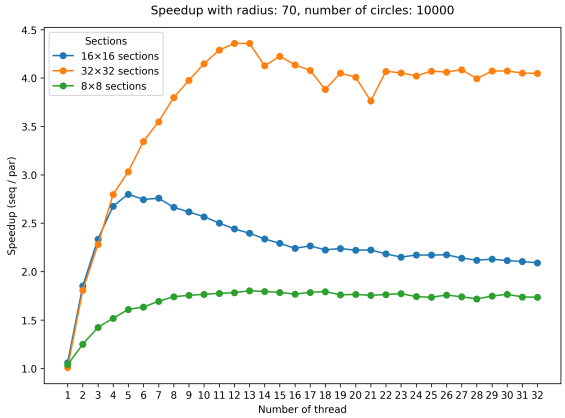

Figure 7. Speedup with 10000 circles of radius not bigger than 70 pixels as the number of sections varies.

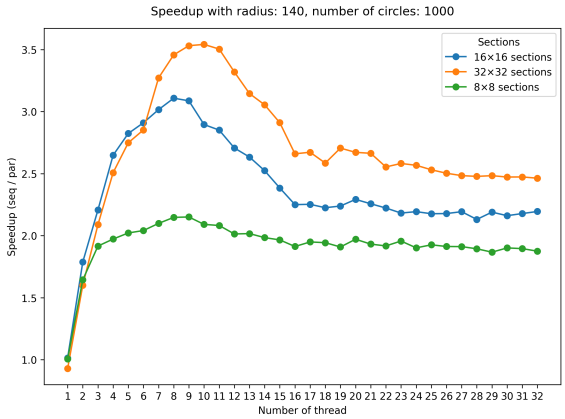

Figure 8. Speedup with 1000 circles of radius not bigger than 140 pixels as the number of sections varies.

Lastly, we analyzed time spent to compute each phase. In Figure 9 we can notice that mapping circles to pixels is the most computationally intensive step and that time taken by mapping circle to section is negligible with respect to all other phases. We can also notice from Figure 11 and 12 that speedup gained with smaller circle is higher than the one obtained with larger ones. That's because with big circle laying on more than one section we are processing the same circle multiple times.

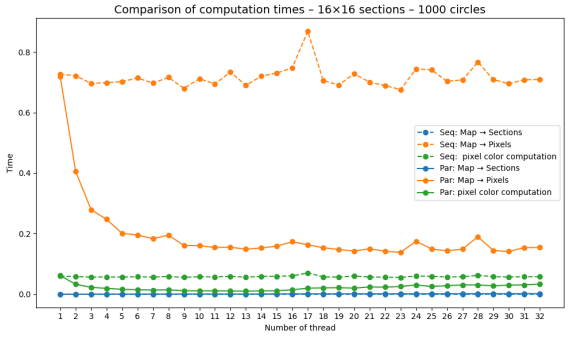

Figure 9. Times taken to compute each step with 1000 circles, 16x16 section and radius not bigger than 70 pixels, in seconds

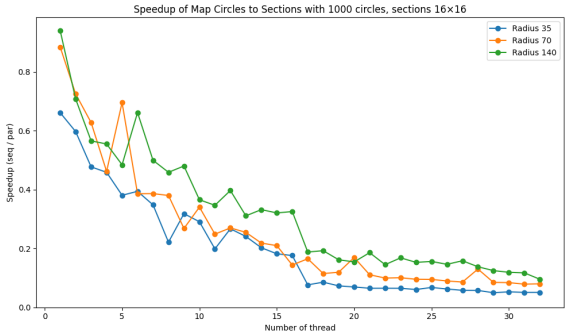

Figure 10. Mapping circle to sections speedup with 1000 circles, 16x16 section.

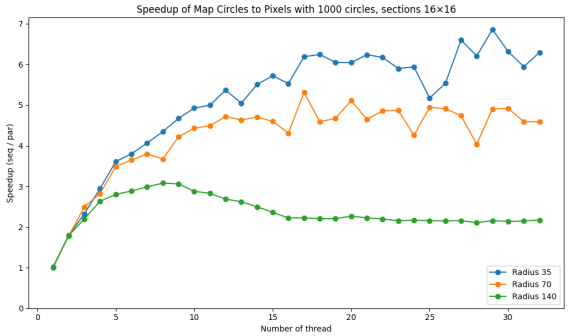

Figure 11. Mapping circle to pixel speedup with 1000 circles, 16x16 section.

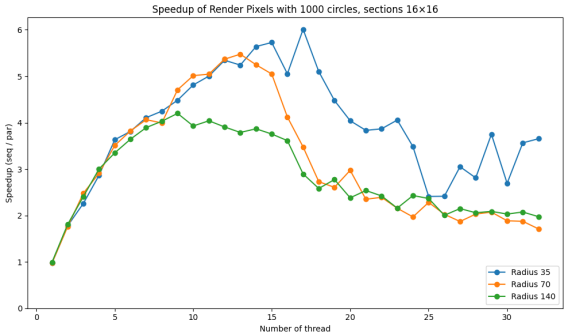

Figure 12. Compute pixel color level speedup with 1000 circles, 16x16 section.

We can also notice that the mapping circles to sections step is a critical one, since we have parallel computation times higher than sequential ones. To understand why this happens, we deeply analyzed this step and found out (as can be observed in Figure 13) that this step isn't complex enough to justify parallelization. Indeed the thread spawning time itself is equal or exceeds total sequential time.

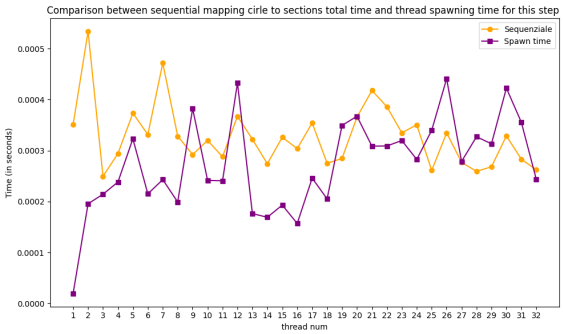

Figure 13. Comparison between total sequential time and thread spawning time

4.1. Profiling

Lastly we used AMDuProf to profile the parallel implementation. We used as test case an image with 10000 circles and radius not bigger than 70, varying number of sections.

This tests refers to case shown in Figure 7, so we profiled runs with 8 and 16 threads and with 5 and 12 threads (peak of speedup curve with sections 16x16 and 32x32, peak for 8x8 curve is 8).

In Figure 14 we can observe that, rendering an image with 8x8 section, if we parallelize the code using 8 threads, we have the maximum time spent using all thread simultaneously, minimizing time wasted. This can be observed even better in Figure 15, where we can see in gray inactive regions. Using 8 thread we can minimize time spent with threads waiting and consequentially gain the better speedup; by increasing the number of threads,

we lose the benefits of increased parallelization due to the many inactive regions.

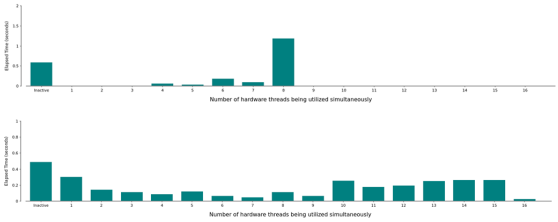

Figure 14. Number of thread used simultaneously with sections 8x8. Above: sample test made using 8 threads, below: sample test made using 16 threads.

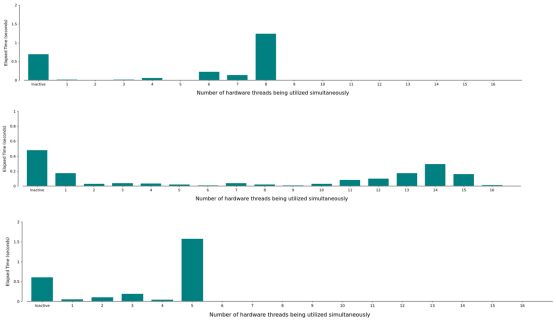

Figure 16. Number of thread used simultaneously with sections 16x16. Above: sample test with 8 threads, middle: sample test with 16 threads, below: sample test with 5 threads.

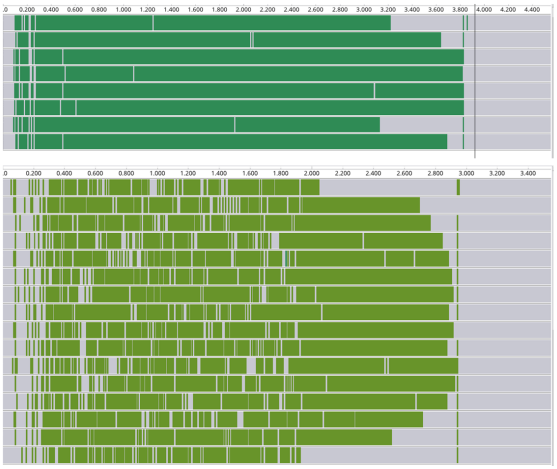

Figure 15. Core occupation, in gray inactivity time.

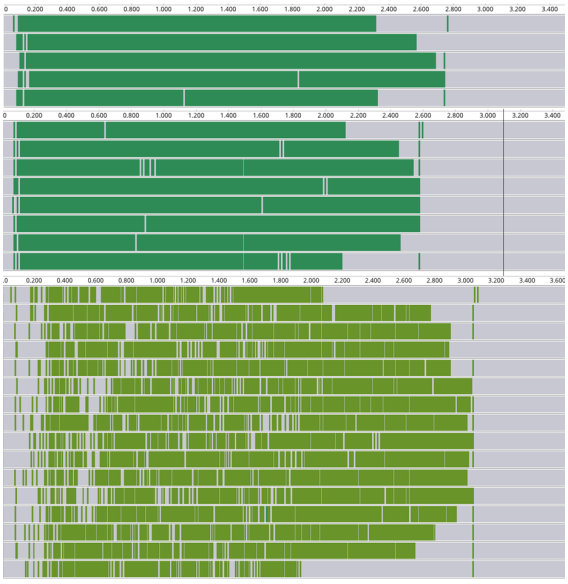

Figure 17. Core occupation, in gray inactivity time.

We can see similar behavior in Figure 16 and Figure 17; in this case we are analyzing rendered image with 16x16 section and we can see that we are in the best scenario with 5 threads. Similarly, in this case, if we increase the number of threads to 16, for most of the time, we have a number of physical threads running simultaneously well below 16. We can observe that we have optimal results also for 8 thread, since in this case, even if we must deal with more inactive regions, we have the shortest simulation time.

Lastly, with 32x32 sections, we can benefit from the use of a large number of threads. With 12 threads, we can spend most of the time using all threads simultaneously, minimize inactive regions, but we can see that, like in the previous case, we gain optimal results also with 8 threads.

To really see the benefit of using an higher number of threads, we must analyze a more challenging problem. In Figure 20 and Figure 21 we can observe simultaneously used threads while rendering 32x32 image with 10000 circles of radius not bigger than 140. As shown in Figure 8, we can now obtain good speedup with 16 threads.

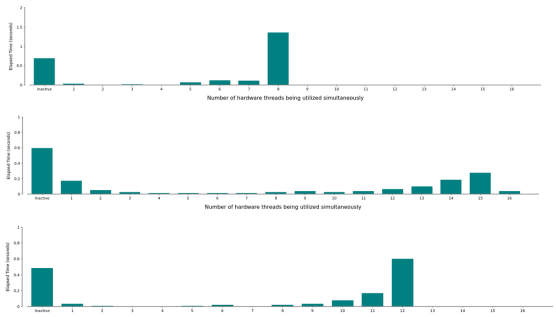

Figure 18. Number of thread used simultaneously with sections 32x32. Above: sample test with 8 threads, middle: sample test with 16 threads, below: sample test with 12 threads.

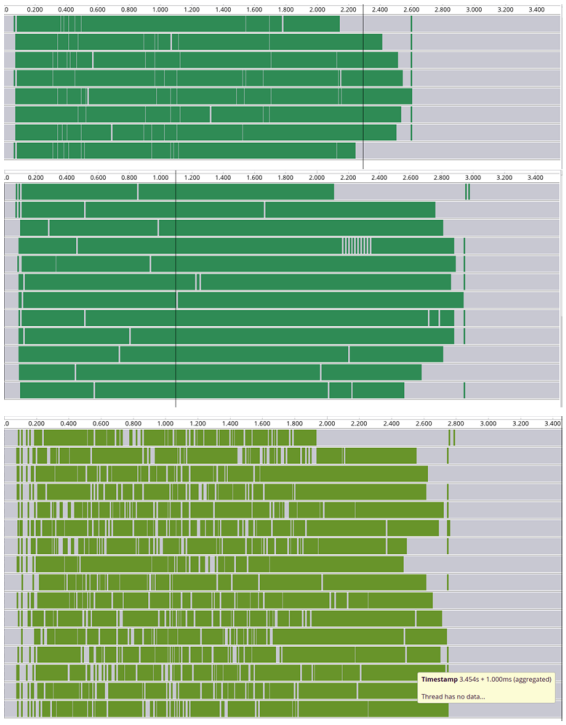

Figure 19. Core occupation, in gray inactivity time.

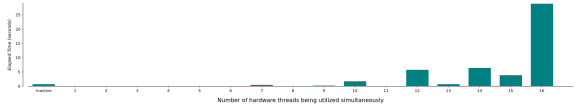

Figure 20. Number of thread used simultaneously with sections 32x32.

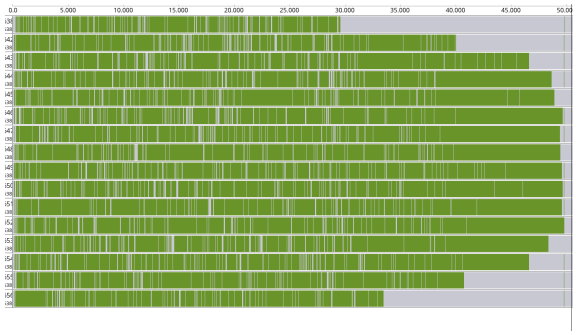

Figure 21. Core occupation, in gray inactivity time.

5. Conclusions

In this paper, we have implemented a sequential and a parallel version of a circle renderer, analyzing how this case of study benefits of parallelism. We built a pipeline in which we tried to emphasize parallelization limiting synchronization to the initial part. We have seen how this can bring to a good speedup in some cases, and also how in other setting the synchronous nature of this problem represents a limit from a parallel point of view.