PC-2023/24 Renderer Project using OpenMP

Elizaveta Mochalova
elizaveta.mochalova@edu.unifi.it
The University of Florence

Abstract

This project implements a Circle-Renderer that draws semi-transparent circles based on a randomly generated set of coordinates. The implementation then gets executed in parallel and in sequence to compare the speedup for the two modes.

Future Distribution Permission

1. Introduction

This project focuses on the implementation and performance evaluation of a Circle-Renderer. The primary objective of the project is to explore the efficiency and speedup achieved through sequential and parallel execution modes.

This Circle-Renderer generates circles using SFML graphics library based on a set of coordinates, producing an image that can be saved to compare the results later on. To analyze the computational performance, the implementation is executed in two distinct modes: parallel and sequential, based on the user's input. The parallel execution mode uses OpenMP library to optimize some blocks of code where it is possible to do so, while the sequential mode follows a traditional, step-by-step approach.

The project tries to compare the speedup achieved by the parallel execution against its sequential counterpart. This evaluation serves to provide insights into the potential benefits of parallelization for programs of similar scope and that use SFML library, which can create some challenges for the parallelization techniques.

1.1. SFML library

The program utilized SFML, Simple and Fast Multimedia Library to generate a static array of circles, each with distinct positions, colors, and sizes.

The *sf::RenderWindow* class created the graphical window, allowing for the display of the circles. The window's size is based on the current monitor video mode settings, so it can easily adjust to a different monitor.

The *sf::CircleShape* class enabled the creation and manipulation of circular shapes within the window with parameters set differently for each particular shape using the *createCircle* method. Since SFML is a 2D library that by default does not support three dimensional shapes, the circle was instead defined as a struct *CircleData* to be able to keep track of the z coordinate of each circle.

The *sf::Clock* class allowed for the measurement of elapsed time, allowing to measure the program's execution duration and to be able to calculate the speedup afterwards.

1.2. Random number generator

As stated above the renderer creates circles randomly, it does so using *std::random_device*, it serves as a source of non-deterministic random numbers, in this case as a seed for a pseudo-random number generator *std::mt19937*. The different distribution used are:

- Position coordinates

The program generates a coordinate set (x, y, z) that later gets used to create the random circle. Since the window size is determined by the computer current video mode, for x and y variables 150 gets subtracted to prevent having circles that are out ob bounds. For the z coordinate a maximum value of 50 is set arbitrarily, this value at the drawing step determines which shapes are in front.

- Radius

  The program generates a radius for the circles of value between 5 and 50.

- Colour in RGB

  The random generator also decides the colour of the circles in RGB colour system, that means each value is between 0 and 255. It is later assigned to the circle that needs to be created.

```
int colour[3];
colour[0] = colorDist(genCol);
colour[1] = colorDist(genCol);
colour[2] = colorDist(genCol);
```

1.3. Parallelization using OpenMP

For this project OpenMP directives were used to parallelize the execution of the program. The loop iterations are divided among the available threads, potentially improving performance on systems with multiple processors or cores.

When the execution reaches a parallel section (marked by *omp pragma*), this directive will cause slave threads to form. Each thread executes the parallel section of the code independently.

- Creation of circles The loop that iterates over the creation of circles was parallelized. Here is the simplified version of the code:

```
#pragma omp parallel for
for (int i = 0; i < NoS; i++){
    shapes[i] = createCircle();}
```

Number of circles	PC	Server
10	0.78	...
100	1.04	...
500	1.13	...
1.000	1.08	...
10.000	1.91	...
20.000	2.19	...

Table 1. Speed up values comparison

- Drawing In this case, it enables parallel execution of the loop that draws the circles.

```
#pragma omp for
for (int i = 0; i < NoS; i++){
    window.draw(shapes[i].circle);}
```

It is important to keep in mind that the effectiveness of parallelization depends on various factors, including the nature of the workload and the underlying hardware. Because of this different methods of execution were tested to find the optimal solution for this specific case.

2. Results and conclusions

This program evaluates the performance of various numbers of circles, that depend on vector *differentNoS* and produces an image as demonstrated in figs. 2 to 5

The achieved speedup factors for different configurations are presented in the following Table 1. This table also demonstrates the speedup for a PC and a for DINFO server.

The observed speedup factors reveal predictable trends in the algorithm's performance. The parallelization exhibits noticeable improvement for a bigger number of circles, with a peak speedup of 2.19 achieved when rendering 20.000 circles concurrently. However, the efficiency of parallelization goes down for smaller workloads, as indicated by the suboptimal speedup factors for fewer circles.

The changes in speedup across different configurations can be attributed to factors such as the workload distribution, overhead times, and the hardware architecture. The diminishing returns for smaller numbers of circles suggest that the

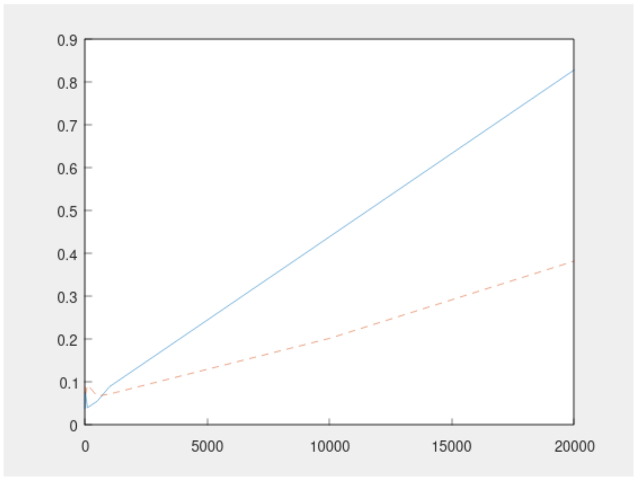

Figure 1. Graph of speed up for parallel and sequential execution.

parallelization strategy may incur additional overhead in such scenarios, this in fact can be seen in figure 1.

These findings underscore the importance of carefully selecting parallelization strategies based on the nature and scale of the workload.

In conclusion, the presented results highlight the trade-offs and nuances associated with a program of this type using SFML graphics library.

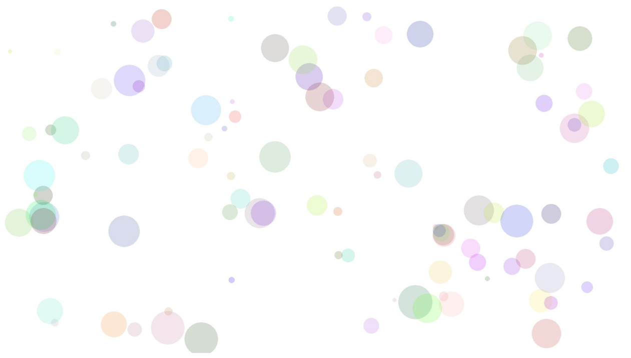

Figure 2. 100 randomly generated circles in parallel.

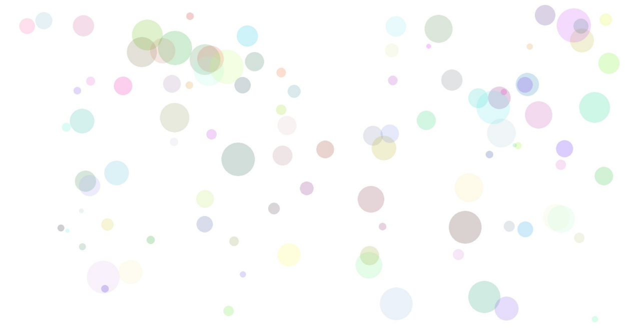

Figure 3. 100 randomly generated circles in sequence.

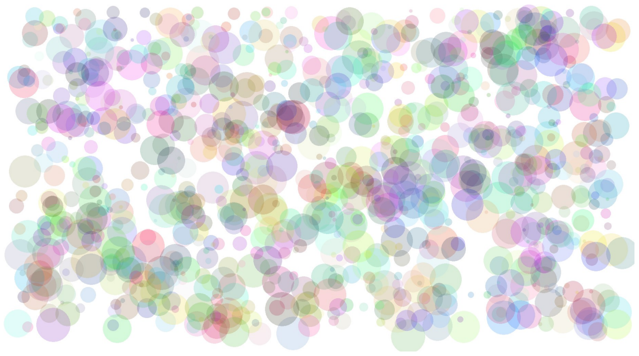

Figure 4. 1000 randomly generated circles.

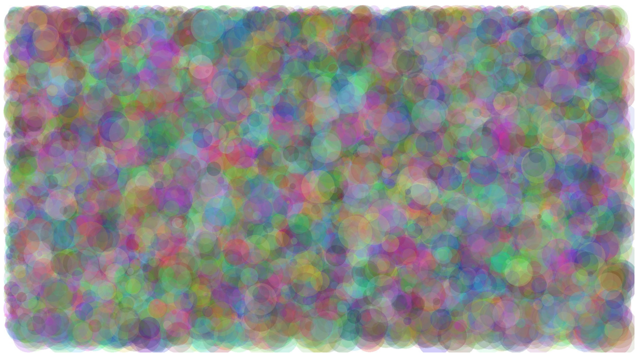

Figure 5. 20.000 randomly generated circles.