Parallel Computing 2023-24.
Renderer project using OpenMP

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

Abstract

This project studies the implementation and performance evaluation of a Circle Renderer using the SFML graphics library, comparing sequential and parallel execution modes. The program generates random circles, varying in position, radius, and color, and visualizes them in a 2D window. Parallel execution uses OpenMP to optimize performance by distributing workload across multiple threads. The study evaluates the speedup achieved through parallelization, considering factors like workload distribution, synchronization overhead, and hardware architecture. Results indicate that while parallelization offers potential benefits, the effectiveness varies depending on the complexity of bitmap construction and system resources. Optimizations such as thread management, load balancing, and vectorization are suggested to further enhance performance.

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 an 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: sequential and parallel. 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 compares the speedup achieved by the parallel execution with its sequential counterpart. This evaluation serves to provide insights into the potential benefits of parallelization for programs of similar scope that use the SFML library, which can create some challenges for 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 size is fixed to 700x700 pixels.

```
int imgWidth = 700;
int imgHeight = 700;
sf::RenderWindow windowSeq
    (sf::VideoMode(imgWidth,
    imgHeight));
```

The *sf::CircleShape* class enabled the creation and manipulation of circular shapes within the window with parameters set differently for each particular shape using a custom *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.

```
sf::CircleShape circle;
int depth;
```

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,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 distributions used are:

- Position coordinates.

  The program generates a coordinate set (x, y, z) that is used to create the random circle. Since the window size is fixed, the maximum value for the x coordinate is "imgWidth - 2 * offsetCenter", where offsetCenter is a variable set to 20. The same applies to y, but its maximum value is "imgHeight - 2 * offsetCenter". This prevents circles from being drawn out of bounds. The z coordinate has a maximum value of 8, set arbitrarily. During the drawing step, this value determines which shapes appear in front.

  ```
  int xPosition = posXDist(gen);
  int yPosition = posYDist(gen);
  int zPosition = posZDist(gen);
  ```

- Radius.

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

  ```
  int radius = radiusDist(gen);
  ```

- Colour in RGB.

  The random generator also determines the color of the circles using the RGBA color system, meaning each value ranges from 0 to 255. The generated color is then assigned to the circle that is being created.

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

1.3. Sequential execution

This section of the code executes sequential rendering process for drawing circles in an SFML window. A *sf::RenderWindow* is created with a fixed size to display the rendered circles. The circles are sorted in descending order based on their depth to ensure correct rendering order, since deeper circles need to be drawn first. After sorting, each circle is drawn onto the window. A *sf::Texture texture* is then created and updated to capture the window's content, saving it as an image file.

The elapsed rendering time is recorded and stored in *seqTimes* for performance analysis.

1.4. Parallelization using OpenMP

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

A *std::map Bitmap* is created to store bitmaps corresponding to different depth levels.

Multiple OMP directives were used in the program to accelerate the performance.

- Bitmap Initialization.

  #pragma omp parallel for directive enables multiple threads to concurrently initialize the *bitmapsWithDepth* map, creating a different BitMap for each depth level. This parallelization ensures efficient memory allocation and avoids performance bottlenecks from sequential initialization.

- Rendering Circles in Parallel.

#pragma omp for for whene depth values are iterated over, each set of circles at a given depth is processed. For each circle the relevant pixels within its bounding box are checked using *pixelInCircle()* method. If a pixel has not been colored yet, it is assigned the circle's color. Otherwise, colors are blended using *blendColorsBitmaps()*. OpenMP speeds up this per-pixel operation by allowing multiple threads to process different depths in parallel.

- Final Image Composition.

  #pragma omp for uses for loop that combines individual depth bitmaps into a final image *bitmapFinal*. Each pixel is checked across all depth layers (starting from the deepest) and blended accordingly. If a pixel is initially transparent, it takes the first non-transparent color found in the depth layers. Otherwise, the colors are blended. Important note is that here *#pragma omp critical* ensures that only one thread at a time modifies *bitmapFinal*, preventing race conditions or unexpected results.

After processing, the final bitmap is converted into an *sf::Image*, which is displayed in an *sf::RenderWindow*. Upon closing the window, the final rendered image is saved to a file. The execution time is recorded and stored in *parTimes* for later performance comparison.

By leveraging OpenMP, this implementation significantly accelerates circle rendering compared to the sequential approach, particularly when handling a large number of overlapping shapes. The key optimizations come from parallelizing bitmap initialization, depth-based circle processing, and final pixel blending, ensuring an efficient multi-threaded execution.

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.

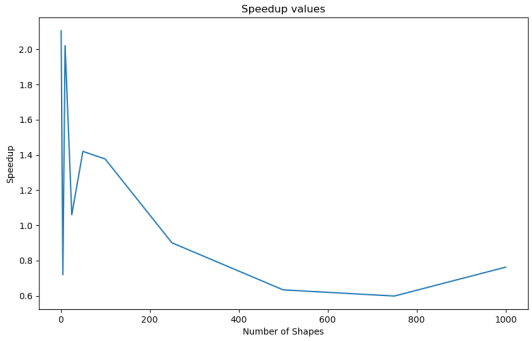


Figure 1: Speedup graph.

2. Results and conclusions

This program evaluates the performance of rendering different numbers of circles, determined by the vector *differentNoS*, which includes values such as 1, 5, 10, 25, 50, 100, 250, 500, 750, 1000. The resulting images are shown in Figures 2a, 2b, 3a, and 3b.

The achieved speedup factors for different configurations are presented in the following table ..... Variations in speedup can be attributed to several factors, including workload distribution, synchronization overhead, and hardware architecture. In many cases, the sequential implementation outperforms the parallel version. This is primarily due to the more computationally intensive method used to construct the bitmap in the parallel section, leading to increased overhead. This trend is evident in Figure 1.

To improve performance, several optimizations can be considered:

- Thread management.

  The speedup potential depends on the number of available threads. Increasing the number of threads up to the system's hardware limit can improve performance, but excessive threads may introduce scheduling overhead.

- Better load balancing.

  Ensuring an even distribution of workload among threads can prevent bottlenecks, so in

put case it depends on how many circles there are per layer.

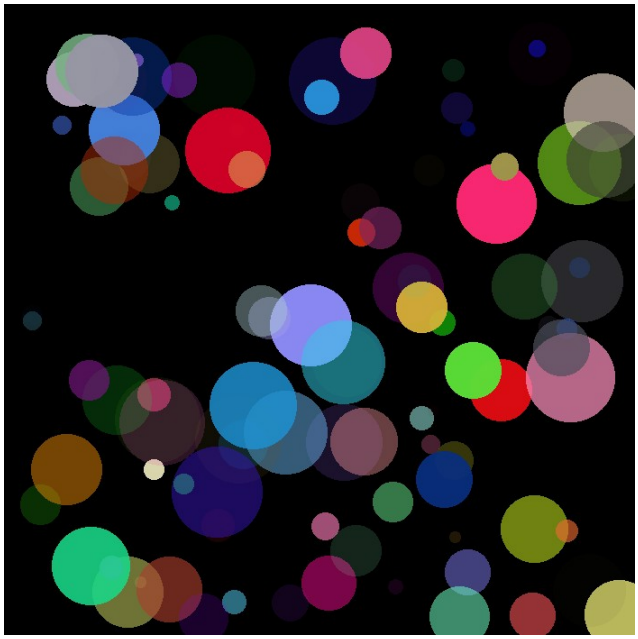
- Vectorization and SIMD:

  Exploiting modern CPU vectorization capabilities can speed up computations.

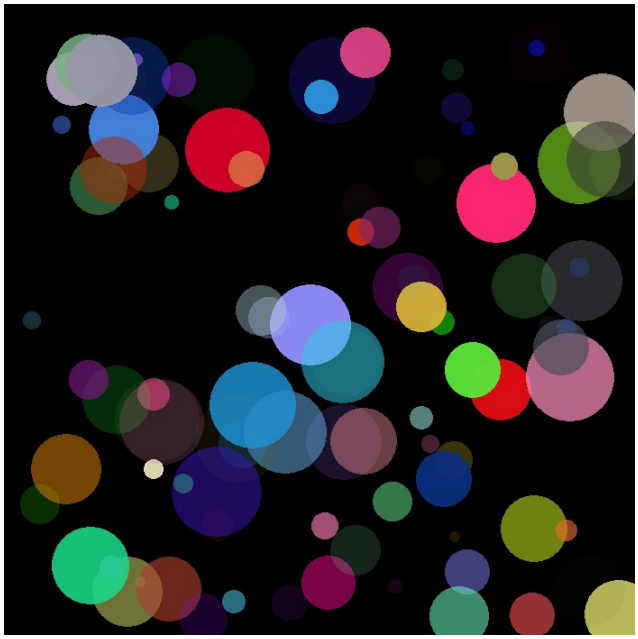
- Asynchronous processing:

  Using task-based parallelism instead of traditional multi-threading could help optimize performance by overlapping computation and rendering, but it is not always possible.

In conclusion, the results highlight the trade-offs associated with parallelization in a graphics-based program using the SFML library. While parallel execution has potential advantages, its effectiveness depends on implementation details, hardware constraints, and the overhead introduced by synchronization and bitmap construction.

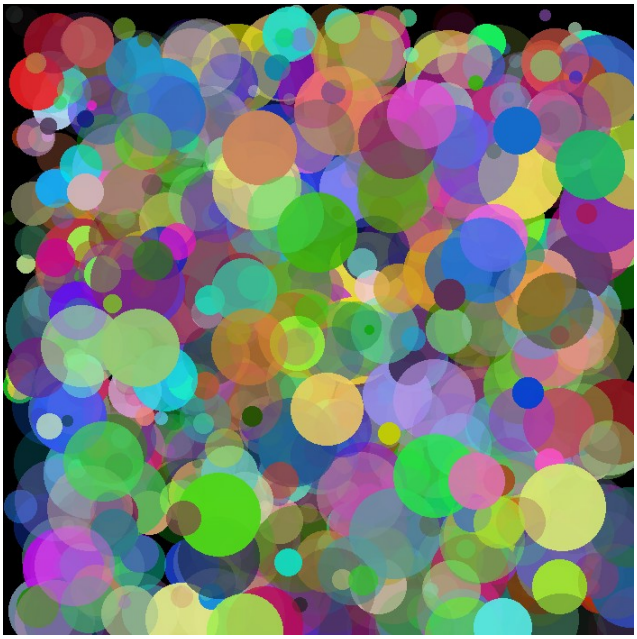


(a) 100 randomly generated circles in parallel.

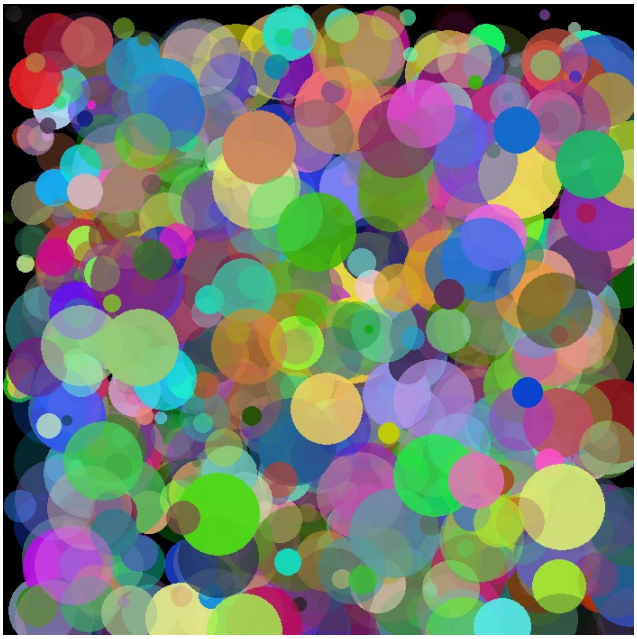


(b) 100 randomly generated circles in sequence.

Figure 2: Comparison of parallel vs. sequential generation of circles for 100 shapes.



(a) 1000 randomly generated circles in parallel.



(b) 1000 randomly generated circles in sequence.

Figure 3: Comparison of parallel vs. sequential generation of circles for 1000 shapes.