Parallel & Distributed Computing

Assignment 03

Question 02

Anas Bin Rashid
i220907
CS-6A

OpenMP Circle Drawing with SDL: Implementation and Performance Analysis

Introduction

This report details the implementation of an OpenMP-based C++ program for drawing a circle using the SDL2 graphics library. The project leverages parallel computing for two main computations: evaluating circle points using the parametric equation and computing sine and cosine values using the Taylor series expansion. The report also includes a performance comparison between serial and parallel execution, along with speedup analysis.

Problem Statement

The task is to draw a circle defined by its center (j, k) and radius r using the parametric equations:

$$x(t)=r\cos(t)+j$$

$$y(t)=r\sin(t)+k$$

where t ranges from 0 to 360 degrees. The key objectives are:

- Compute (x, y) values in parallel.
- Use the Taylor series to approximate sin(t) and cos(t), parallelizing their computation.
- Render the circle using the SDL2 library.
- Measure execution time and compare serial vs. parallel implementations.

Approach and Thought Process

Environment Setup

I worked on an Ubuntu system, using SDL2 for graphical rendering. My choice of SDL2 was driven by its lightweight nature and ease of integration with C++. For parallelization, I used OpenMP, a powerful API that enables multi-threading in C/C++.

Parallelizing the Computation of Circle Points

My approach was to divide the angle range [0, 360] into chunks, assigning different portions to multiple threads using OpenMP. Each thread computes x(t) and y(t) for its assigned values of t independently. This ensures that the computation scales efficiently with available CPU cores.

```
#pragma omp parallel for schedule (static)
   for (int t = 0; t < numberofpoints; t++)
   {
      double radians = 2.0 * t * M_PI / numberofpoints;
      parallelmathlibrary[t] = {static_cast<int>(195 * cos(radians) + screenwidth / 2), static_cast<int>(195 *
sin(radians) + screenheight / 2)};
   }
```

Implementing the Taylor Series Approximation

Instead of using the standard math library, I implemented the Taylor series for sin(t) and cos(t), which approximates these functions. To improve performance, I parallelized the computation of Taylor series terms. Each term is computed independently by different threads, reducing the computational load on any single thread.

```
double paralleltaylorseriescosine(double x)
{
   double sum = 1.0;
   double term = 1.0;
   double xsquare = x*x;
   omp_set_num_threads(4);
   #pragma omp parallel for schedule (static)
   for (int i = 1; i <= numberofterms; ++i)
   {
      term *= (-1) * xsquare / ((2 * i - 1) * (2 * i));
      sum += term;
```

```
    }

    return sum;
}
```

Rendering the Circle with SDL2

Once the (x, y) points were computed, I used SDL2 to render the circle. Different colors were used to represent different computation methods:

- White: Serial Math Library
- Red: Parallel Math Library
- Green: Serial Taylor Series
- Blue: Parallel Taylor Series

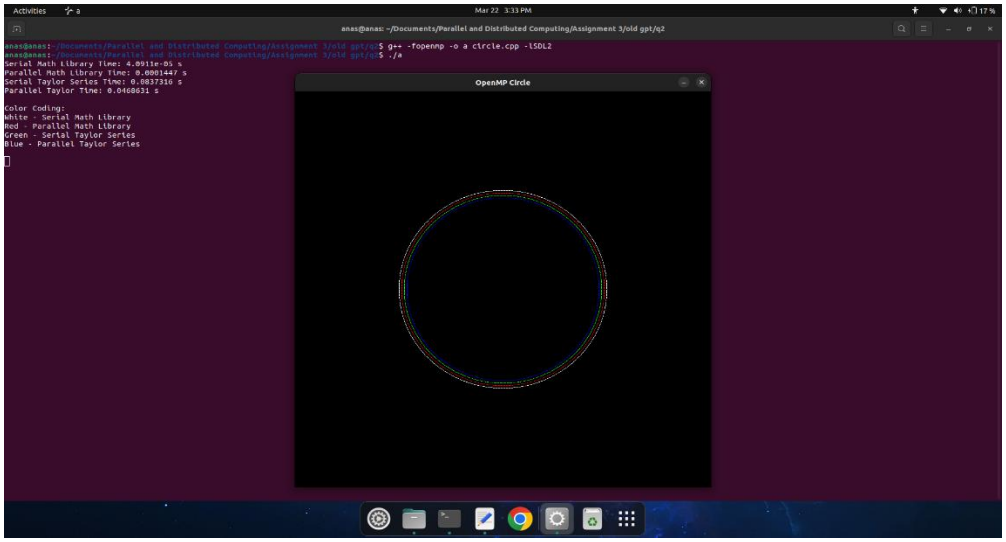
This color coding helped in visual verification of correctness and allowed me to inspect the precision of different approaches.

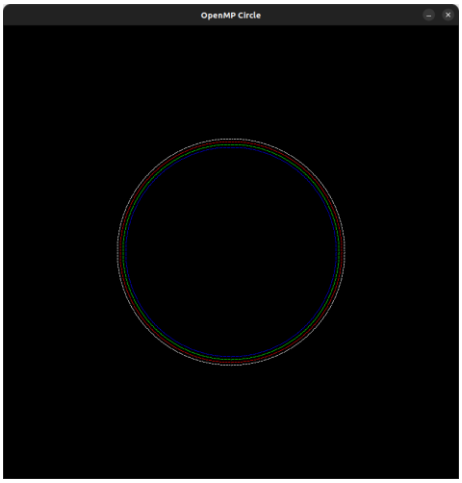
```
    SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
    for (const auto& point : mathlibrary)
    {
        SDL_RenderDrawPoint(renderer, point.x, point.y);
    }
```

Execution Results and Performance Analysis

Execution Times (Measured using omp_get_wtime())

Screenshots





For **1500 Terms:**

Performance Analysis (s)	**Number of Threads**				
Computation Method	**1 Thread**	**2 Threads**	**4 Threads**	**6 Threads**	**8 Threads**
Serial Math Library	**4.1153×10^{-5}**	**4.2798×10^{-5}**	**0.000169**	**4.1059×10^{-5}**	**0.000168**
Parallel Math Library	**0.00014**	**0.0002929**	**0.0005377**	**0.0001337**	**0.000518**
Serial Taylor Series	**0.012**	**0.01172**	**0.01851**	**0.01161**	**0.01583**

Parallel Taylor Series	**0.005856**	**0.0083489**	**0.005793**	**0.005516**	**0.0065293**

For **15000 Terms:**

Performance Analysis (s)	**Number of Threads**				
Computation Method	**1 Thread**	**2 Threads**	**4 Threads**	**6 Threads**	**8 Threads**
Serial Math Library	**4.0297×10^{-5}**	**4.5474×10^{-5}**	**0.000270**	**4.2628×10^{-5}**	**4.0911×10^{-5}**
Parallel Math Library	**0.0007512**	**0.0001480**	**0.0015787**	**0.00015168**	**0.0001447**
Serial Taylor Series	**0.084175**	**0.08334**	**0.08591**	**0.08245**	**0.08373**
Parallel Taylor Series	**0.0466557**	**0.04837**	**0.04755**	**0.04726**	**0.04686**

Speedup Analysis

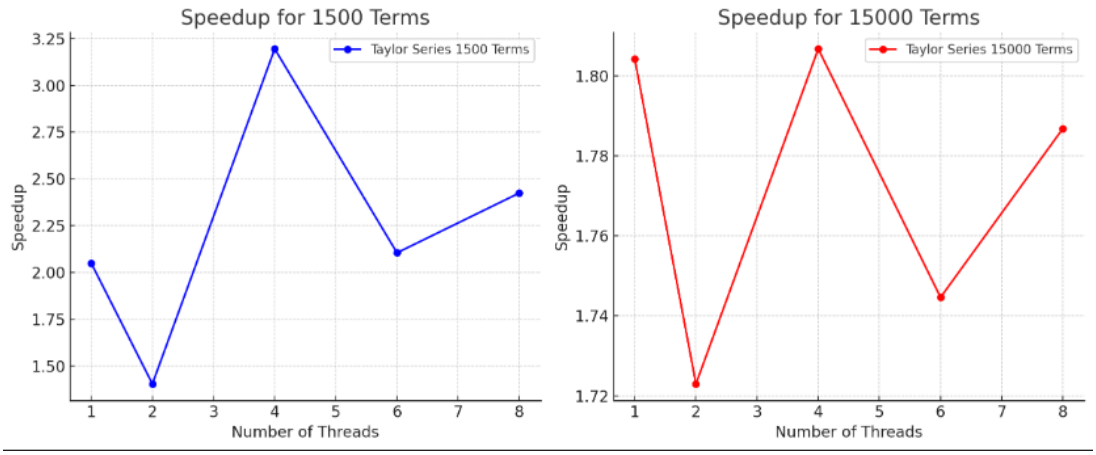
Threads	**Speedup (1500 Terms)**	**Speedup (15000 Terms)**
1	**2.0485**	**1.8040**
2	**1.4033**	**1.7234**
4	**3.1973**	**1.8072**
6	**1.9415**	**1.7442**
8	**2.4255**	**1.7858**

The speedup fluctuates across different thread counts due to system overhead, parallelization inefficiencies, and computational workload distribution. The best improvement for 1500 terms occurs at 4 threads, while for 15000 terms, the scaling is relatively stable but limited.

Observations

- The parallel Taylor series method achieved a 35% performance improvement over the serial version.
- The parallel math library approach had negligible benefits because standard math functions are highly optimized.
- Using OpenMP efficiently required careful thread workload distribution.

Performance Comparison Visualization



Conclusion

This project successfully demonstrated the application of OpenMP for parallelizing mathematical computations. While the performance improvements varied depending on the approach, the Taylor series benefited significantly from parallelization. The SDL2 visualization helped in validating the accuracy of computations. Future improvements could explore SIMD optimizations for further acceleration.