13X007: Assignment #3

Due on 08.11.2023

Parallelism

UNIVERSITÉ
DE GENÈVE

CHRISTOFOROU Anthony

Abstract

This report delves into the implementation and analysis of a parallelized C++ program, designed for calculating the value of π using the Riemann sum method, with a focus on employing OpenMP for efficient parallel computation. The report elaborates on the theoretical foundation, challenges, performance results, and potential improvements in the context of parallel computing and numerical methods.

Contents

1 Introduction

The calculation of π using numerical methods is a classic problem in computational mathematics. This report presents a parallelized approach using the Riemann sum method in C++, leveraging the OpenMP framework for enhanced computational performance. The objective is to demonstrate efficient parallel computing techniques in approximating mathematical constants and to analyze the scalability and accuracy of such methods.

2 Theoretical Background

2.1 Riemann Sum

$$\int_0^1 f(x) = \lim_{|\Delta x| \to \infty} f(x_i^*)\Delta x \tag{1}$$

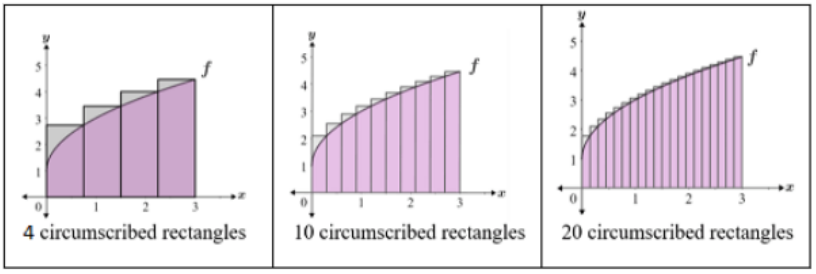


Figure 1: Rieman sum converging to integral

The Riemann sum is a fundamental concept for approximating the definite integral of a function. It involves summing the areas of rectangles under a curve, providing an approximation of the integral. The basic formula for a Riemann sum is:

$$S = \sum_{i=1}^{n} f(x_i)\Delta x$$

where Δx is the width of each sub-interval, and $f(x_i)$ is the function value at a chosen point x_i in the i-th sub-interval.

2.2 Integration of $f(x) = \frac{1}{1+x^2}$

The function $f(x) = \frac{1}{1+x^2}$, when integrated over the interval [0, 1], yields $\pi/4$. Thus, the area under this curve approximates $\pi/4$, and multiplying the result by 4 gives an approximation of π.

3 Implementation

The program calculates the area under the curve $f(x) = \frac{1}{1+x^2}$ by dividing the interval [0, 1] into 10^8 rectangles and summing their areas, using the midpoint of each rectangle to evaluate the function.

3.1 Code Explanation

```
double sum = 0.0;
double step = 1.0 / (double)num_steps;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++) {
    double x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

The code employs OpenMP directives for parallelizing the loop that calculates the Riemann sum. The 'reduction(+:sum)' clause ensures thread-safe addition to the variable 'sum'.

4 Performance Results and Analysis

4.1 Execution Times

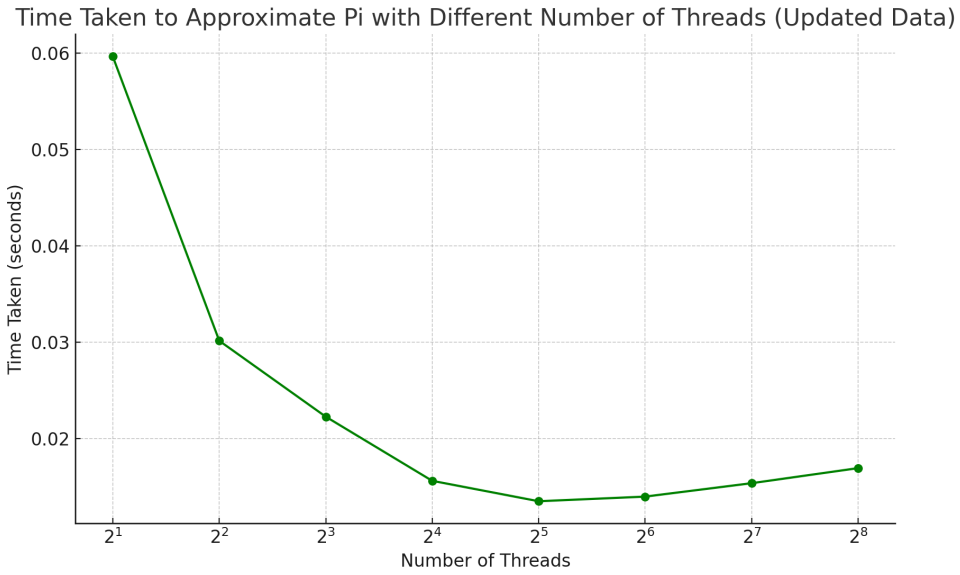
The program's performance was evaluated with varying numbers of threads. The execution times and the calculated values of π were recorded, showing a decrease in computation time with an increase in threads, up to a certain point.

4.2 Output Analysis

The output of the program with different thread counts is as follows:

```
Threads: 2, Time taken: 0.0596318 seconds
Approximation of Pi with 2 threads: 3.14159
Threads: 4, Time taken: 0.030148 seconds
Approximation of Pi with 4 threads: 3.14159
Threads: 8, Time taken: 0.0222797 seconds
Approximation of Pi with 8 threads: 3.14159
Threads: 16, Time taken: 0.0156268 seconds
Approximation of Pi with 16 threads: 3.14159
Threads: 32, Time taken: 0.013509 seconds
Approximation of Pi with 32 threads: 3.14159
Threads: 64, Time taken: 0.0139896 seconds
Approximation of Pi with 64 threads: 3.14159
Threads: 128, Time taken: 0.0153812 seconds
Approximation of Pi with 128 threads: 3.14159
Threads: 256, Time taken: 0.0169438 seconds
Approximation of Pi with 256 threads: 3.14159
```

These results indicate that the parallelized implementation efficiently reduces computation time, especially for lower thread counts. However, beyond a certain number of threads, the improvement in performance becomes less significant.



As the number of processors increases in tasks like Pi approximation, execution time initially drops due to parallelization, where tasks are divided among processors for simultaneous execution. This leads to faster execution when scaling from a single to a moderate number of processors. However, beyond an optimal point, the benefits diminish due to increased overhead from managing more threads. This results in a plateau or slight increase in execution time, demonstrating a balance between parallel processing advantages and the complexities of managing more processors.

5 Challenges and Limitations

The implementation, while effective, faces certain challenges:

- **Scalability**: The diminishing returns in performance improvement with higher thread counts highlight scalability limitations due to overheads in thread management and synchronization. Additionally, the effective distribution of workload (load balancing) becomes increasingly challenging as the number of threads grows, which can lead to inefficiencies and underutilization of resources.
- **Accuracy**: The accuracy of the Riemann sum approach depends on the number of rectangles; more rectangles mean better accuracy but at the cost of increased computation time.
- **Hardware Constraints**: The performance also depends on the hardware capabilities, particularly the number of available processor cores. Additionally, memory constraints can become a bottleneck, especially when dealing with a large number of threads, each requiring its own memory space for computations.

6 Conclusion

This report demonstrates the application of parallel computing in approximating π using the Riemann sum method. By employing OpenMP, the program achieves significant improvements in

computation time, showcasing the potential of parallel programming in numerical methods. Future work could explore alternative approaches and optimizations to overcome scalability and accuracy limitations.