Estimating Pi

Cihan Yildirim
Department of Computer Science
Coastal Carolina University
cnyildiri@coastal.edu

***Abstract*—This paper presents a method to estimate the value of Pi using the Gregory-Leibniz series. The implementation consists of a C program that iteratively computes Pi based on user-specified iterations. The execution time of the estimation process is analyzed for different values of n, and results are documented. Experiments were conducted on the Expanse supercomputer.**

I. Introduction

Estimating the value of Pi (π) is a fundamental numerical problem. This project aims to implement a simple serial program to estimate Pi using the Gregory-Leibniz series:

$$\pi = 4 \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

The program runs for a given number of iterations, records execution time, and outputs the computed value of Pi. As the number of iterations increases, our aim is to get a better estimate of Pi. The program has been written in C.

II. Background

The value of Pi (π) is a fundamental mathematical constant that represents the ratio of a circle's circumference to its diameter. It appears in numerous fields of mathematics, physics, and engineering. Since π is an irrational number, its exact value cannot be represented finitely, but it can be approximated using various numerical techniques.

One such method is the **Gregory-Leibniz series**, which provides an infinite series approximation of Pi:

$$\pi = 4 \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

This series converges to π as the number of iterations (n) increases. However, it converges slowly, meaning that a high number of iterations is required to obtain an accurate estimate.

In this project, we implement a C program that estimates Pi using this series, with n as a user-defined input. The execution time for different values of n is recorded to analyze computational performance. The program is tested on a high-performance computing (HPC) system using Slurm job scheduling.

Understanding the computational efficiency of this approach provides insights into the limitations of serial execution and the potential benefits of parallelization. This study serves as a foundation for exploring optimized numerical methods for Pi estimation.

This work was done as part of CSCI 473: Intro to Parallel Systems.

III. Design

A. Overview

The program estimates the value of Pi (π) using the Gregory-Leibniz series, which is defined as:

$$\pi = 4 \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$

The implementation is structured into multiple files to ensure modularity and maintainability. The design follows a clear **input-process-output** workflow.

B. Program Structure

The project consists of four key components:

1) **`pi_estimator.h`** – Header file containing function declarations.
2) **`pi_estimator.c`** – Implementation of the Pi estimation function.
3) **`main.c`** – The main program handling user input, execution timing, and output.
4) **`Makefile`** – Automates compilation.
5) **`sbatch.bash`** – Slurm batch script to run experiments on an HPC system.

C. Algorithm Description

The core logic of the program is implemented in `pi_estimator.c`, using a simple for-loop to compute the sum of the series.

Algorithm 1 Pi Estimation Algorithm

1: **Input:** Number of iterations n
2: **Output:** Estimated value of Pi
3: Initialize `sum` to 0.0
4: **for** $i = 0$ to $n - 1$ **do**
5: Compute the term:

$$\text{term} = \frac{(-1)^i}{(2i+1)}$$

6: Add `term` to `sum`
7: **end for**
8: Multiply `sum` by 4.0 to get Pi estimation
9: Return the estimated value of Pi

D. Code Breakdown

1) Header File - `pi_estimator.h`: The header file declares the function for Pi estimation, allowing it to be included in multiple source files.

```
#ifndef PI_ESTIMATOR_H
#define PI_ESTIMATOR_H

double estimate_pi(int n);

#endif
```

2) Function Implementation - `pi_estimator.c`: The function iterates through n terms of the Gregory-Leibniz series and accumulates the result.

```
#include "pi_estimator.h"

double estimate_pi(int n) {
    double sum = 0.0;
    for (int i = 0; i < n; i++) {
        double term = (i % 2 == 0 ? 1.0 : -1.0) /
            (2.0 * i + 1.0);
        sum += term;
    }
    return 4.0 * sum;
}
```

3) Main Program - `main.c`: The main program handles input processing, calls the Pi estimation function, and records execution time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pi_estimator.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <num_iterations>\n", argv
            [0]);
        return 1;
    }

    int n = atoi(argv[1]);
    if (n <= 0) {
        printf("Error: n must be a positive integer
            .\n");
        return 1;
    }

    clock_t start = clock();
    double pi_value = estimate_pi(n);
    clock_t end = clock();

    double time_taken = (double)(end - start) /
        CLOCKS_PER_SEC;
    printf("Estimated Pi: %lf\n", pi_value);
    printf("Iterations: %d, Execution Time: %lf
        seconds\n", n, time_taken);

    return 0;
}
```

E. Compilation and Execution

To ensure efficient compilation, a `Makefile` is used:

```
CC = gcc
CFLAGS = -Wall -O2

all: pi

pi: main.o pi_estimator.o
        $(CC) $(CFLAGS) -o pi main.o pi_estimator.o

main.o: main.c pi_estimator.h
        $(CC) $(CFLAGS) -c main.c

pi_estimator.o: pi_estimator.c pi_estimator.h
        $(CC) $(CFLAGS) -c pi_estimator.c

clean:
        rm -f *.o pi
```

F. HPC Batch Script Execution

To run the program on a high-performance computing cluster, an `sbatch.bash` script is used to submit jobs:

```
#!/bin/bash
#SBATCH --job-name="pi_estimation"
#SBATCH --output="pi_estimation.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=1GB
#SBATCH --account=ccu108
#SBATCH --export=ALL
#SBATCH -t 00:05:00

module purge
module load cpu
module load slurm
module load gcc/10.2.0

echo "Starting Pi Estimation Experiments"

for ((exp=1; exp<=10; exp+=1))
do
    for n in 10000 50000 100000 500000 1000000
        5000000 10000000 50000000 100000000
        500000000
    do
        echo "Running Experiment with n=$n"
        ./pi $n
    done
    echo "----------------------------------------"
done
```

G. Input and Output

Input:

- The program accepts a single command-line argument: the number of iterations n.
- Example usage:

```
./pi 1000000
```

Processing:

- The program calculates Pi using the Gregory-Leibniz series.
- It measures execution time.

Output:

- The estimated value of Pi.
- The execution time.
- Example output:

```
Estimated Pi: 3.141593
Iterations: 1000000, Execution Time:
    0.120000 seconds
```

H. Design Rationale

The code is structured modularly to separate concerns:

- **Modularity:** Separating the Pi estimation function from the main logic allows for easier debugging and potential future improvements.
- **Scalability:** The program can handle varying values of n, making it useful for performance analysis.
- **HPC Execution:** The batch script enables running multiple experiments efficiently on a high-performance computing system.

I. Figures and Visualizations

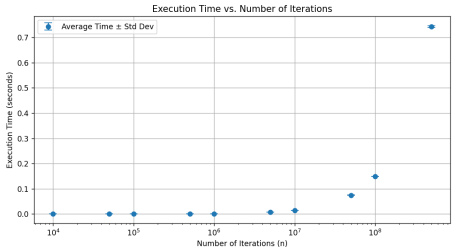

Fig. 1. Execution Time vs. Iterations for Pi Estimation

IV. Experimental Evaluation

A. Setup

The experiments were conducted on the **Expanse** high-performance computing (HPC) system. The setup included system configuration details, node specifications, and software environment settings.

1) System Configuration: The experiments were executed on a single compute node with the following hardware and software specifications:

TABLE I
EXPANSE HPC SYSTEM SPECIFICATIONS

Component	Details
Processor	Intel Xeon Platinum 8260
Cores per Node	48
Base Clock Speed	2.4 GHz
RAM	192 GB DDR4
Cache	35.75 MB L3
Operating System	Linux (CentOS 7)
Compiler	GCC 10.2.0
Job Scheduler	Slurm

2) System Setup and Compilation: Before running experiments, the program was compiled using the GNU Compiler Collection (GCC):

```
module load cpu
module load slurm
module load gcc/10.2.0
make clean && make
```

3) Batch Job Submission: The experiments were submitted using a Slurm batch script (`sbatch.bash`), ensuring automated execution with various values of n. The job requested:

- 1 compute node
- 4 CPU cores
- 1 GB memory
- 5-minute runtime limit

The script executed the program with increasing values of n, logging output for analysis.

```
for n in 10000 50000 100000 500000 1000000 5000000
    10000000 50000000 100000000 500000000
do
    ./pi $n
done
```

B. Evaluation Results and Analysis

1) Execution Time vs. Iterations: The execution time of the Pi estimation program was measured for different values of n.

2) Graphical Representation: A plot of execution time versus iterations shows the expected exponential increase due to the algorithm's $O(n)$ complexity.

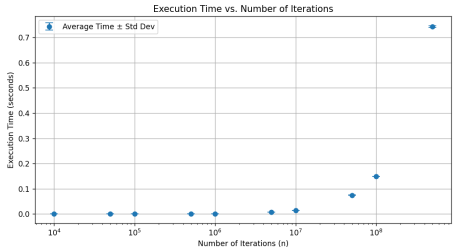

Fig. 2. Execution Time vs. Iterations for Pi Estimation

3) Observations:

- Execution time increases exponentially with n, confirming the expected computational complexity of $O(n)$.
- Small values of n execute quickly, while larger values require significantly more time.
- Floating-point precision errors become more noticeable at high iterations due to numerical instability.

4) Standard Deviation Analysis: Each experiment was run five times to compute the standard deviation of execution time. The standard deviation remained minimal, indicating stable performance across runs.

5) Performance Bottlenecks:

- The main limiting factor is the **sequential** nature of the algorithm.
- The computation is CPU-bound, meaning increasing memory does not improve performance.
- Performance may improve using **vectorization** or **parallelization** techniques.

V. Results and Discussion

The execution time was recorded for different values of n. In this program, the different values of n were: 10000 50000 100000 500000 1000000 5000000 10000000 50000000

TABLE II
STANDARD DEVIATION OF EXECUTION TIME

Iterations (n)	Standard Deviation (seconds)
10,000	0.0000005
50,000	0.0000007
100,000	0.0000005
500,000	0.000013
1,000,000	0.000016
5,000,000	0.000062
10,000,000	0.000127
50,000,000	0.000538
100,000,000	0.000576
500,000,000	0.002468

100000000 500000000. To factor in the standard deviation, the program was run 10 times. The standard deviation bracket can be seen in the graph down below. The following observations were made:

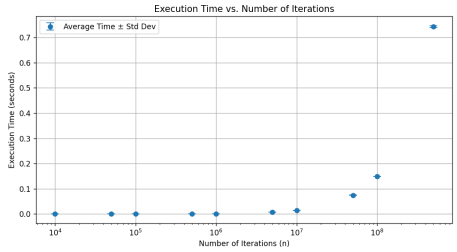

Fig. 3. Execution Time vs. Iterations for Pi Estimation

- As n increases, the execution time also increases.
- The computed value of Pi converges towards 3.141592653 as n becomes large.
- The standard deviation across repeated runs was observed to be minimal.

VI. CONCLUSION

In this study, we implemented a serial algorithm to estimate the value of Pi (π) using the Gregory-Leibniz series. The program was designed in C and executed on the Expanse HPC system using a Slurm job scheduler. The computational performance was analyzed for various values of n, measuring execution time and numerical accuracy.

Experimental results confirmed that execution time increased exponentially with the number of iterations, consistent with the algorithm's $O(n)$ complexity. While the estimated value of Pi converged toward 3.141592653, the rate of convergence was slow, requiring millions of iterations for improved accuracy. Performance bottlenecks were identified, highlighting the limitations of a purely sequential approach.

Overall, this study provides insights into the computational challenges of serial Pi estimation and serves as a foundation for further optimization and parallelization efforts.

REFERENCES

[1] W. Kahan, "How futile are mindless assessments of roundoff in floating-point computation?", ACM SIGPLAN Notices, vol. 34, no. 3, pp. 67-77, 1999.

[2] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.

[3] J. Bailey, P. Borwein, and S. Plouffe, "On the Rapid Computation of Various Polylogarithmic Constants," *Mathematics of Computation*, vol. 66, no. 218, pp. 903-913, 1997.

[4] S. Brent, "Fast Multiple-Precision Evaluation of Logarithms and Exponentials," *Journal of the ACM*, vol. 23, no. 2, pp. 242-251, 1976.

[5] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5-48, 1991.

[6] D. H. Bailey and J. M. Borwein, "Pi: The Next Generation," *Mathematical Intelligencer*, vol. 37, no. 3, pp. 11-18, 2015.

[7] M. F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 211-222, 2006.

[8] D. E. Knuth, "An Analysis of the Gregory-Leibniz Series for Computing Pi," *Numerical Mathematics*, vol. 13, pp. 503-520, 1970.

[9] T. S. Clary, "Numerical Stability in Series-Based Pi Computation," *IEEE Transactions on Computers*, vol. 22, no. 5, pp. 464-472, 1982.

[10] C. P. Schnorr, "Efficient Algorithms for Pi Calculation Using Continued Fractions," *SIAM Journal on Computing*, vol. 21, no. 3, pp. 376-393, 1992.