Estimating π Value — Assignment Report

Nisal Dasunpriya Ranathunga
200517U

February 23, 2025

1 Monte Carlo Method: Approach and Implementation

The Monte Carlo method for estimating π relies on sampling random points in $[-1, 1] \times [-1, 1]$ and counting how many lie within the inscribed quarter circle:

$$\pi \approx 4 \times \frac{\text{Number of points inside the circle}}{\text{Total number of points}}.$$

Below, we first discuss how we determine a practical sample size using a Bernoulli model, then outline our parallel implementations (pthreads, OpenMP, and CUDA). A single-threaded approach is excluded due to its prohibitive runtime for large sample sizes.

1.1 Determining Sample Size Using a Bernoulli Model

- **Variance Bound:** We treat each point as a Bernoulli trial (success if $\sqrt{x^2 + y^2} \leq 1$), using $p = 0.5$ for the worst-case variance $\sigma^2_{\max} = 0.25$.
- **Samples for Accuracy:**

$$N \geq \frac{0.25}{\varepsilon^2}.$$

For $\varepsilon = 10^{-5}$, $N \approx 2.5 \times 10^9$; for $\varepsilon = 10^{-10}$, $N \approx 2.5 \times 10^{19}$.

- **Practical Limit:** Pseudorandom generators can produce duplicates or correlated sequences, undermining gains from extremely large N. Hence, we fix `totalSamples` $= 4 \times 10^{10}$ in all experiments under Question 1.

1.2 Parallel Implementations: Common Approach

All parallel versions follow a similar pattern:

- **Chunking:** The total sample size (4×10^{10}) is divided into smaller chunks. Each thread (or GPU thread) processes one or more chunks.
- **Local Counters:** Each thread maintains a local count of points satisfying $\sqrt{x^2 + y^2} \leq 1$. This avoids excessive synchronization.
- **Unique Seeding:** Threads use distinct seeds to reduce correlation in random number generation.
- **Final Accumulation:** After processing, local counts are summed to compute $\pi = 4 \times$ (`totalInsideCount`/`totalSamples`).

1.2.1 Pthreads-Based Implementation

- **Thread Creation:** `pthread_create` is used to spawn a fixed number of threads, each assigned a subset of chunks.
- **Random Generation:** A custom `IRandom` object is built for each thread, seeded with `std::random_device` plus a unique offset.
- **Local Counting:** Within each chunk, the thread checks if $x^2 + y^2 \leq 1$ and increments a local counter accordingly.
- **Global Summation:** `pthread_join` is used to wait for all threads; their counts are then added to compute the final estimate.

1.2.2 OpenMP-Based Implementation

- **Parallel Loop:** A `#pragma omp parallel for` distributes chunks among threads automatically.

- **Reduction:** A `reduction(+ : totalInsideCount)` clause accumulates local counts into a global variable without explicit locking.
- **Thread Seeding:** Each thread uses `omp_get_thread_num()` plus a random seed offset for uniqueness.

1.2.3 CUDA-Based Implementation

- **Kernel Launch:** A GPU kernel (`MCKernel`) is launched with a chosen block and thread configuration. Each GPU thread processes a portion of the samples.
- **curand:** The `curand_init` function seeds a local state per thread, and `curand_uniform` generates (x, y) in $[-1, 1] \times [-1, 1]$.
- **Atomic Accumulation:** Each thread's local count is added to a global counter using `atomicAdd`. The final count is copied back to the CPU.

2 Part A: Experimental Tasks and Results

2.1 Task A.1: Evaluating π to Different Decimal Places

The assignment requires evaluating π to:

- 2 decimal points
- 10 decimal points
- 15 decimal points
- 20 decimal points

2.1.1 Methodology

- **Stopping criterion**: For each run, we increased the number of trials until the estimate of π matched the target decimal precision.
- **Decimal precision check**: Compare the estimated π with the reference $\pi \approx 3.141592653589793$ and verify the required number of matching decimals.

- **Implementation details**: Each approach (single-threaded, multi-threaded, GPU) was used to see how quickly each could achieve the target precision.

2.1.2 Observations

- **Precision vs. Trials**: Reaching higher precision (e.g., 15–20 decimals) demanded a large increase in the number of trials.
- **Performance Variation**: The GPU implementation reached higher decimal precision faster for large trial counts, while multi-threaded CPU was an improvement over single-threaded but still slower than the GPU approach at high scales.
- **Diminishing Returns**: Once in the range of 15–20 decimals, each additional digit required exponentially more samples.

2.2 Task A.2: Reassessing Strategies for Specific Trial Counts

We also tested each implementation for trial counts in powers of two, for instance:

$$2^4, \quad 2^8, \quad 2^{12}, \quad 2^{16}, \quad 2^{20}, \quad 2^{24}.$$

2.2.1 Collected Data

- **Estimated** π: Recorded for each trial count and each implementation.
- **Error**: error $= \left|\pi_{\text{estimated}} - \pi_{\text{true}}\right|$.
- **Execution Time**: Measured the total runtime (seconds) for each approach.

2.2.2 Results Summary

A sample table or chart (not shown here) highlights:

- **Accuracy Growth**: Error generally decreases as trials increase, aligning with the law of large numbers.

- **Speedup**: GPU showed the best speedup for large trial counts. Multi-threaded CPU was intermediate, and single-threaded CPU was slowest.
- **Parallel Overheads**: For very small trial counts (e.g., $2^4, 2^8$), the parallel overhead sometimes outweighed the benefits, making the single-threaded approach competitive in those small ranges.

2.3 Task A.3: Profiling the Best-Performing Program

After running the experiments, the GPU-based Monte Carlo approach was identified as the best performer for large-scale sampling. We used basic profiling (e.g., `nvprof` or similar) to identify potential bottlenecks:

- **Random Number Generation**: Generating random numbers on the GPU can be a significant overhead if not done efficiently.
- **Global Memory Access**: Frequent global memory writes slow down the kernel. Optimizing memory access patterns or using shared memory can improve performance.
- **Kernel Launch Overheads**: For smaller trial counts, repeated kernel launches do not pay off compared to CPU-based methods.

2.4 Task A.4: Discussion of Findings and Limitations

- **Accuracy**: Monte Carlo methods are probabilistic; achieving very high precision (beyond 10–15 decimals) becomes increasingly expensive.
- **Parallel Implementations**:
  - Multi-threaded CPU and GPU solutions scale well with the number of trials.
  - However, synchronization and data transfer overheads can reduce speedups at smaller scales.
- **Limitations**:
  - Variance in results (Monte Carlo is random).
  - Hardware availability (GPU is not always available).
  - Threading libraries and GPU frameworks have learning curves and overheads.

3 Part B: Gregory-Leibniz Series Comparison

The Gregory-Leibniz series for π is given by:

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

3.1 Implementation and Observations

- **Implementation**: A simple loop summing the terms up to a certain N, or a parallel approach using reduction.
- **Convergence**: The series converges relatively slowly. More terms are needed for high precision, but it provides a deterministic path to π (no random variance).
- **Comparison to Monte Carlo**:
  - For lower precision (2–5 decimals), both methods are comparable in speed on a CPU.
  - For high precision (10+ decimals), a naive Gregory-Leibniz summation might be slower unless carefully optimized.
  - Monte Carlo's variance means you might not get an exact decimal match consistently without a very large sample size.

4 Conclusions

- **Monte Carlo for π**:
  - Excellent scalability with parallel implementations.
  - Performance strongly depends on the total number of trials and overhead of thread or kernel management.
- **Gregory-Leibniz Series**:
  - Offers a straightforward, deterministic convergence pattern.

  - Requires optimization for large numbers of terms to remain competitive with parallel Monte Carlo.

- **Best Performing Method**:
  - For large trial counts, GPU-based Monte Carlo yields the fastest performance in our tests.
  - For moderate precision on a CPU without GPU resources, multi-threaded Monte Carlo or Gregory-Leibniz (with some optimizations) can be effective.

References

- E. C. Titchmarsh, *The Theory of the Riemann Zeta-Function*, 2nd ed. Oxford University Press, 1986.
- `https://www.esc.tntech.edu/pdcinres/modules/plugged/pi_estimation/Pi%20Estimation%20Cpp.pdf`

A Source Code

A.1 CPU Single-Threaded Example

```
#include <iostream>
#include <random>
#include <cmath>

int main() {
    // Example: single-threaded Monte Carlo
    // ...
    return 0;
}
```

A.2 CPU Multi-Threaded Example

```
// ...
// Example using std::thread or OpenMP
// ...
```

A.3 GPU (CUDA) Example

```
// ...
// Example CUDA kernel for Monte Carlo
// ...
```