Parallel Monte Carlo Estimation of π

DVA336 – Parallel Systems

Barbulescu Bogdan-Catalin

1 Introduction

1.1 The Monte Carlo Method

The Monte Carlo method originated in the 1940s, when scientists at Los Alamos used random simulations to study complex phenomena during nuclear research. It relies on pseudo-random numbers (generated from a starting seed) to simulate statistical models or evaluate complex probability [1]. As described in the seminal work by Metropolis and Ulam, "The method is, essentially, a statistical approach to the study of differential equations, or more generally, of integro-differential equations that occur in various branches of the natural sciences" [2].

In this project, we apply the Monte Carlo method to estimate the value of π and implement a parallel version using OpenMP. The basic idea is to generate random points (x, y) inside a square of side $2r$ centered at the origin, then count how many points fall inside a circle of radius r inscribed in the square. The computed ratio of points inside the circle to the total number of points is used to estimate π (Figure 1).

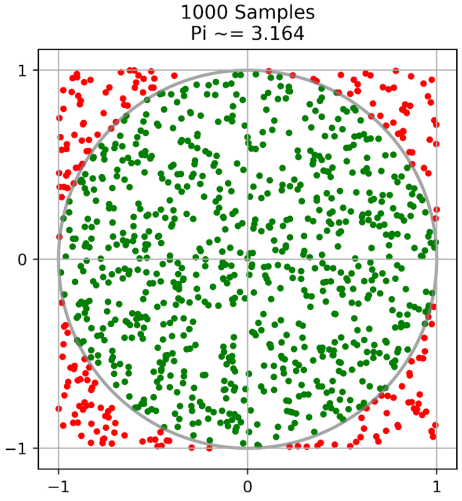

Figure 1: Square and inscribed circle used for the Monte Carlo estimation of π.

1.2 Sequential Implementation

The following code represents a sequential implementation of the algorithm:

```
long inside = 0;

srand(time(nullptr));

for (long i = 0; i < N; i++) {
    double x = rand() / (double)RAND_MAX;
    double y = rand() / (double)RAND_MAX;
    if (x*x + y*y <= 1.0)
        inside++;
}

double pi_estimate = 4.0 * inside / N;
double error = std::abs(pi_estimate - M_PI);
```

Explanation:

- `inside` counts the points that fall inside the circle.
- `N` refers to the total number of points generated.

- π is estimated as $4 \cdot \texttt{inside}/N$.
- `error` is the absolute difference between the estimated and the true value of π.
- The `srand` function ensures that each execution generates a different sequence of pseudo-random numbers.

2 Parallel Approach

The Monte Carlo algorithm relies on generating random points and counting how many fall inside the circle. Each point is generated independently, meaning that iterations of the main loop have no data dependencies. This makes the algorithm well suited for parallel execution using OpenMP.

The only shared variable is `inside`, which accumulates the number of points inside the circle. By applying an OpenMP `reduction` clause, each thread maintains a private counter, avoiding race conditions and eliminating the need for explicit synchronization.

2.1 Implementation

Each thread creates a separate pseudo-random number sequence using a private seed based on the thread ID and the current UNIX time. The iterations of the loop are evenly distributed across threads, and a reduction operation ensures the final result is correct.

```
#pragma omp parallel reduction(+:inside) num_threads(
   NUM_THREADS)
{
    unsigned int seed = omp_get_thread_num() + time(
       nullptr);

    #pragma omp for
    for (long i = 0; i < N; i++) {
        double x = rand_r(&seed) / (double)RAND_MAX;
        double y = rand_r(&seed) / (double)RAND_MAX;
        if (x*x + y*y <= 1.0)
            inside++;
    }
}
```

3 Asymptotical Analysis, strong & weak scaling

3.1 Sequential Time Complexity

The sequential algorithm iterates over N points, performing a constant number of operations per iteration. Therefore, the sequential runtime is:

$$T_{\text{seq}}(N) = \alpha \cdot N$$

where α represents the time required for one iteration.

3.2 Parallel Time Complexity

In the parallel version using P threads, each thread executes approximately N/P iterations. The total runtime can be expressed as:

$$T_{\text{par}}(N, P) = \frac{T_{\text{seq}}(N)}{P} + T_{\text{red}}(P) + T_{\text{th}}(P)$$

where:

- $T_{\text{red}}(P)$ is the cost of the reduction operation.
- $T_{\text{th}}(P)$ accounts for thread creation and scheduling overhead.

For large N, the overhead terms become negligible, yielding:

$$T_{\text{par}}(N, P) \approx \frac{\alpha N}{P}$$

3.3 Strong Scaling

Strong scaling measures how the execution time decreases when the number of threads increases while keeping the problem size N fixed. The ideal speedup is:

$$S_{\text{strong}}(P) = \frac{T_{\text{seq}}(N)}{T_{\text{par}}(N, P)} \approx P$$

3.4 Weak Scaling

Weak scaling measures how the runtime changes when the workload per thread remains constant. Let $N = P \cdot n_0$, where n_0 is the number of points per thread. Then:

$$T_{\text{weak}}(P) = \alpha n_0 + T_{\text{red}}(P) + T_{\text{th}}(P)$$

For sufficiently large N, the runtime remains approximately constant as P increases.

4 Experiments

Experiments were conducted on a system equipped with a 12th Gen Intel Core i5-1235U processor, featuring 6 physical cores and 12 hardware threads through simultaneous multithreading (SMT). All programs were compiled using `g++` with optimization level `-O2`, the C++17 standard, and OpenMP support enabled (`g++ -O2 -std=c++17 -fopenmp`). For each configuration, results were obtained by averaging 10 independent runs to reduce measurement variability.

n	p	$t(s)$	s
10^8	1	0.671	1.00×
10^8	2	0.360	1.88×
10^8	4	0.203	3.30×
10^8	8	0.146	4.59×
10^8	12	0.123	5.47×
10^9	1	6.270	1.00×
10^9	2	3.451	1.81×
10^9	4	1.970	3.18×
10^9	8	1.242	5.04×
10^9	12	0.923	6.79×

Table 1: Experimental results from strong-scalability analysis conducted on small (10^8) and large (10^9) problem sizes. The first column reports the input size n, the second the number of threads p, the third the elapsed time in seconds, and the last column the speedup s relative to the single-thread execution.

Having side by side the output of the two cases of small and large input arrays in Table 1, it is obvious that the overhead (i.e. the extra time and computing resources required to manage the parallel execution which is not spent on the actual task itself) negatively impacts the expected output.

While the execution time decreases as the number of threads increases, the speedup drifts away from the ideal one. For example, with $p = 12$, the ideal speedup would be 12×, but the measured speedup is only 6.79× for the larger input.

Comparing the two datasets, the larger input consistently achieves better scalability than the smaller input, confirming the negative impact of the parallel overhead which is less significant when the input size is larger, as the computation time dominates the overhead costs.

n	p	t_S	t_A	s
10^8	1	0.671	0.671	1.00×
2×10^8	2	1.26	0.667	1.88×
4×10^8	4	2.565	0.875	2.93×
8×10^8	8	5.236	1.034	5.06×
12×10^8	12	7.486	1.138	6.57×

Table 2: Experimental results from weak-scalability analysis. The first column reports the input size n, the second the number of threads p, the third and fourth columns the sequential and parallel execution times in seconds, and the last column the resulting speedup s.

While in Table 2 the input size n was increased proportionally to the number of threads p to maintain a constant workload per core, the results demonstrate excellent scalability at lower thread counts. For instance, when doubling the resources from $p = 1$ to $p = 2$, the execution time remains almost unchanged. However, as the number of threads increases further (to 4, 8 and 12), the execution time t_A begins to grow, reaching 1.138s at $p = 12$.

References

[1] J. McCreary (2001). Various Estimations of π as Demonstrations of the Monte Carlo Method. Tennessee Technological University, No. 2001-4.

[2] N. Metropolis and S. Ulam (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, Vol. 44, No. 247, pp. 335-341.