# Markov Chain Monte Carlo Method & Error, Speedup Analysis


## Krut Patel

### High Performance Computing Project, UMass Dartmouth



**Abstract**

In this project, I implemented the Markov Chain method into our Monte Carlo $\pi$ approximation code and compared the results with the standard Monte Carlo code. Additionally, I made the code parallel using OpenMP and conducted error and efficiency analyses across multiple threads. The primary emphasis will be on evaluating the impact of the number of threads on parallel and efficiency analysis for scalability. All coding is performed in C, and the analysis is conducted using Python.


**1. Introduction to Monte Carlo Methods and MCMC :** Monte Carlo methods and Markov Chain Monte Carlo (MCMC) techniques are powerful computational tools widely employed for simulation, optimization, and statistical inference in various fields.

Monte Carlo methods, named after the famous casino in Monaco, utilize random sampling to approximate numerical results for problems that might be analytically intractable. By generating numerous random samples, Monte Carlo methods provide estimates for complex systems, offering a probabilistic approach to problem-solving. A classic example is the estimation of $\pi$ by randomly placing points within a square and determining their positions relative to a quarter-circle inscribed in that square.

Markov Chain Monte Carlo, on the other hand, introduces a dynamic and sequential aspect to the random sampling process. Markov chains, which undergo transitions between different states with defined probabilities, are leveraged to explore complex probability distributions. MCMC methods involve proposing new states based on the current state and accepting or rejecting them according to a specified criterion. This iterative process creates a Markov chain that eventually converges to the desired distribution. A common application is Bayesian statistics, where MCMC aids in sampling from high-dimensional posterior distributions.

The key distinction between traditional Monte Carlo methods and MCMC lies in the sequential and dependent nature of the latter. While Monte Carlo methods generate independent random samples, MCMC builds a correlated sequence of samples, exploiting the Markov property for more efficient exploration of complex parameter spaces. This dependency allows MCMC to navigate through regions of high probability in a targeted manner, making it particularly effective for problems with intricate and multi-dimensional structures.

In summary, Monte Carlo methods encompass a broad class of algorithms relying on random sampling, while Markov Chain Monte Carlo introduces a sequential, Markov chain-based approach for sampling from complex probability distributions. MCMC's ability to explore dependent samples in a correlated manner distinguishes it from traditional Monte Carlo methods, making it a valuable tool in situations where the underlying structure of the problem is not easily accessible through independent sampling.

**2. Algorithm and Parallel code :** Below is the code block that shows the implementation of the Markov Chain Monte Carlo method.

Listing 1: MCMC Pi Estimation

```
int is_inside_circle(double x, double y)
    {
    return x * x + y * y <= 1.0;
}

double estimate_pi_mcmc(long long int N)
    {
    long long int inside_circle_count =
        0;
    unsigned int seed = (unsigned int)
        time(NULL);

    for (long long int i = 0; i < N; ++i
        ) {
        double x = ((double)rand_r(&seed
            ) / RAND_MAX) * 2.0 - 1.0, y
            = ((double)rand_r(&seed) /
            RAND_MAX) * 2.0 - 1.0;
        inside_circle_count +=
            is_inside_circle(x, y);
    }

    return (inside_circle_count * 4.0) /
        N;
}
```

In the Monte Carlo $\pi$ estimation code, a Markov Chain is implicitly implemented within the loop. Random points in the unit square are sequentially generated, creating a dynamic Markov chain. Each iteration proposes new points, evolving the chain, and acceptance is contingent on the points falling within the unit circle. This process efficiently explores the parameter space, enabling the estimation of $\pi$ by considering the ratio of accepted points, reflecting the underlying Markov Chain Monte Carlo methodology for probabilistic computation.

Listing 2: OpenMP Directives for Parallel Code

```
#pragma omp parallel for reduction(+:
    inside_circle_count) private(seed)
```

The pragma omp parallel directive initiates a parallel region, indicating that the code within this region will be executed concurrently by multiple threads.

The pragma omp for directive parallelizes a loop by distributing its iterations among available threads. Each thread executes a portion of the loop iterations.

The reduction clause specifies a reduction operation (addition in this case) and a variable (inside_circle_count) to store the result. It ensures that each thread maintains a private copy of inside circle count, and the final result is obtained by summing values from all threads.

The private(seed) clause declares that each thread should have its private copy of the seed variable. Without this clause, all threads would share the same seed, leading to non-deterministic behavior due to race conditions.

By adding these OpenMP directives before the for loop in the Markov Chain Monte Carlo code, the loop iterations are distributed among multiple threads. Each thread has a private copy of inside_circle_count and the seed variable, ensuring that calculations within the loop are independent and do not interfere with each other.

Parallelizing the loop in this manner significantly accelerates computation, especially when dealing with a large number of iterations. It allows multiple threads to work concurrently on different parts of the loop, leveraging multi-core processors and enhancing overall performance.

**3. Analysis** Let's first take a look at the error analysis. Figure 1 and 2 show the problem size $N$ versus average error for Markov Chain Monte Carlo and a simple Monte Carlo method for a large number of runs. This allows us to examine the true relationship between the error value and the problem size $N$.

Both the plots show similar results for error estimates. We can clearly see that, as both methods are implementations of the Monte Carlo method, the error value ($E$) is proportional to $N^{-1/2}$.

There was one interesting thing I noticed. Figure 1 presents the error estimates for the Markov Chain Monte Carlo (MCMC) method for $N$ up to $10^{12}$. According to the trend, the error should have decreased, but after $10^{10}$ sample points, the error value seemed to be stuck at $10^{-6}$. This is peculiar, as the error should ideally decrease with an increase in problem size.

Figure 3 shows us how does the number of threads used for the parallel program affect the error estimate. The plot has problem size N on x-axis and error estimate on y-axis. It is clear that the number of threads does not induce fluctuations in the error estimate. Infact, it shows the the code produces more consitant results with increased thread count.

Figure 4 also shows a similar kind of results. We're looking at the effect the problem size has on the error estimate. It can be seen that as the problem size increases, a lot of fluctuations are introduced in the error estimate. Which is expected, since working with a larger number of sample points can produce a range of error value for the same problem size. Even from this plot we can say the the number of threads used has little to no effect on the estimated result.

The next two figures show the scaling tests. figure 5 is strong scaling test and Figure 6 is weak scaling test

Let's look at the scaling tests now. The next and final two figures show the scaling tests. figure 5 is strong scaling test and Figure 6 is weak scaling test. From the strong scaling test, as the threads are increasing, the speedup value is decreasing. Which is kind expected if we assume that the code is not fully parallalized. For thread values up to 10, we can see the speedup is almost perfect. After that, even for a large number of N, it is decreasing. There can be a lot of reasons for that. The quality of the parallel code, the size of the problem, the complexity of the function approximated, etc. From he weak scaling test, we can see that increament in the problem size does shoe increment in speedup for all the number of threads. By analyzing the pot, we can see that the optimal number of threads for my code is 27 threads. That is the most number of threads that gives almost perfect speedup. After that, the speedup decreases rapedly. Infact, we can see from the data we have that it is not possible to get a speedup more then 30 for any number of threads even for the largest problem size I used for this code.

**4. Possible Improvments :** To enhance the code, consider optimizing the random number generation process for improved efficiency. Additionally, exploring advanced parallelization techniques and algorithmic optimizations may further boost performance. Employing a more sophisticated error analysis and refining the convergence behavior of the MCMC algorithm could provide valuable insights. Implementation of a more complex integral could also give better insight in saclability, since $\pi$ approximation is not a very load caring algorithm.

**5. Conclusion :** In conclusion, the implemented Markov Chain Monte Carlo method demonstrates effective parallelization using OpenMP, showcasing notable speedup. The observed plateau in error beyond $10^{12}$ sample points warrants investigation, suggesting potential algorithmic constraints or numerical stability issues. Further refinement of the algorithm, coupled with rigorous testing, is essential for harnessing the full potential of parallelized Monte Carlo simulations in high-performance computing environments. It seems that a very large problem size and a more complex algorithm is required to test the MCMC parallel code for a large nnumber of threads as in this code, it soon becomes pointless to use a large number of threads.

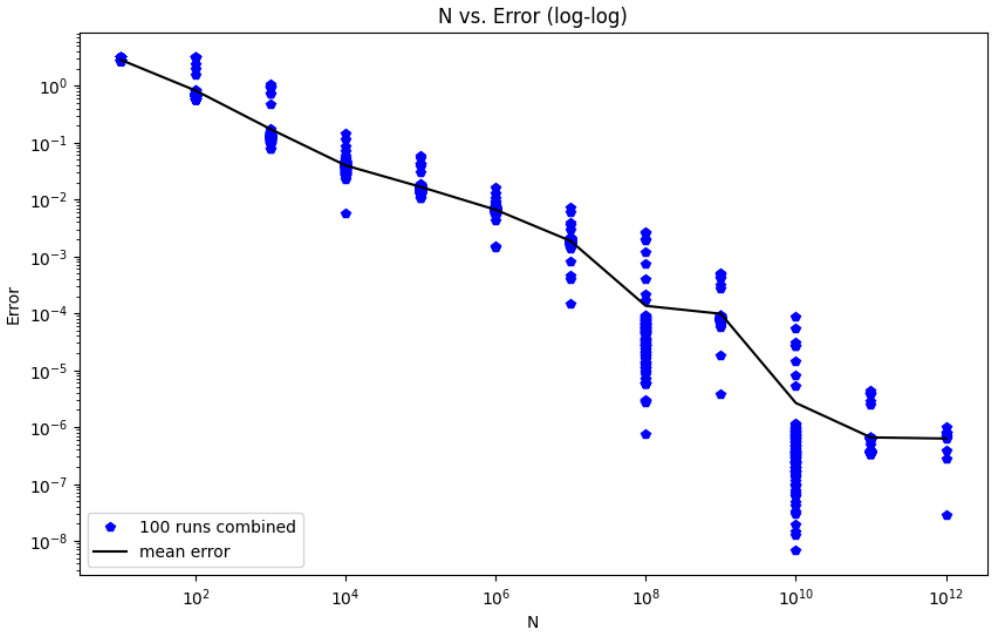


Figure 1: A large number of total runs for increasingly $N$ values versus the average error estimate for the corresponding $N$ value is plotted for the Markov Chain Monte Carlo method.

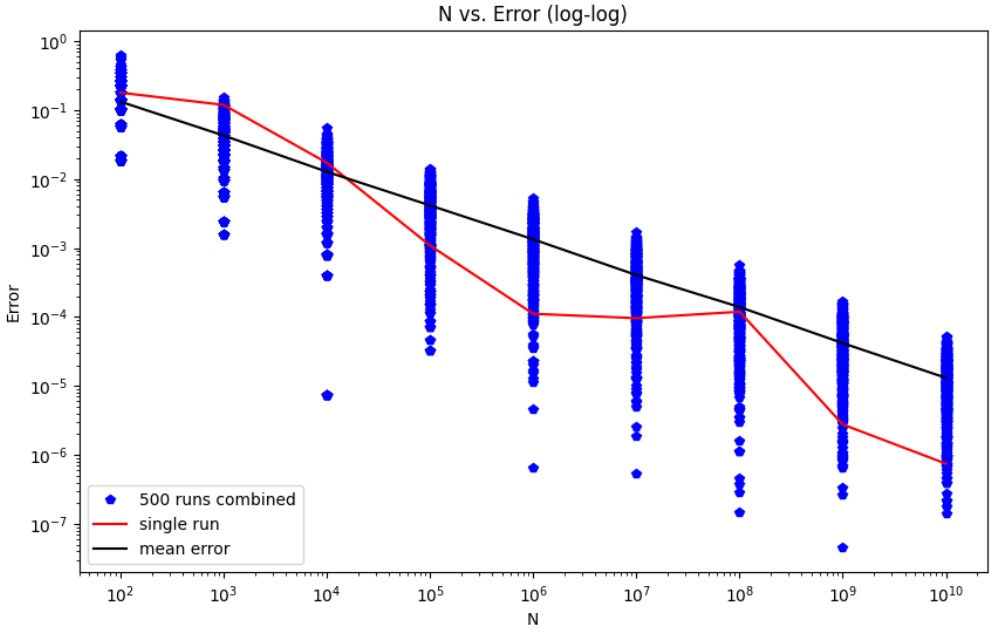


Figure 2: A large number of total runs for increasingly $N$ values versus the average error estimate for the corresponding $N$ value is plotted for a simple Monte Carlo method.

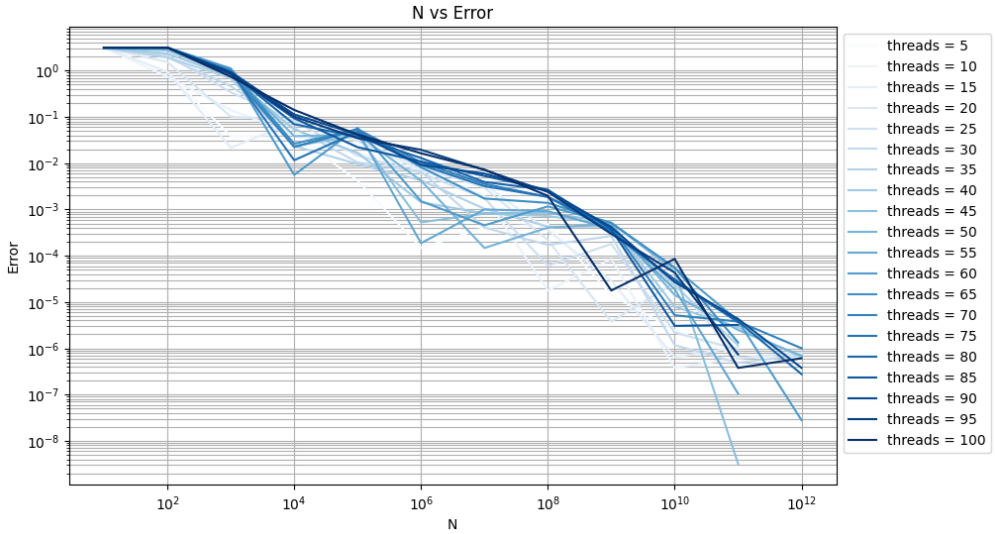


Figure 3: A simple N vs Error plot for different thread values from 5 to 100 to check the effect of increased threads on error estimated.

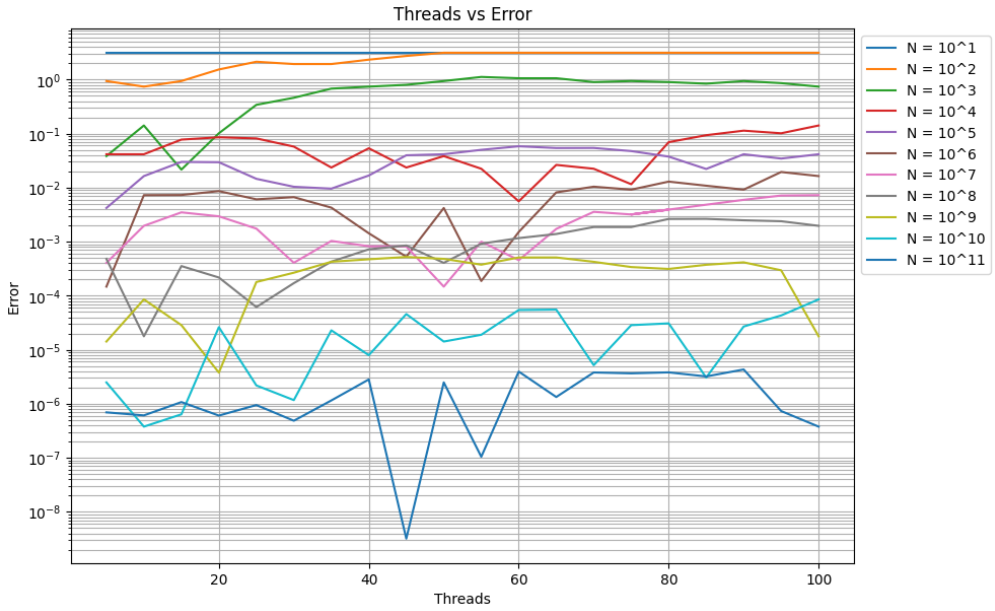


Figure 4: A similar threads vs error plot for different N values to see the effect of N values on the error with respect to the thread count.

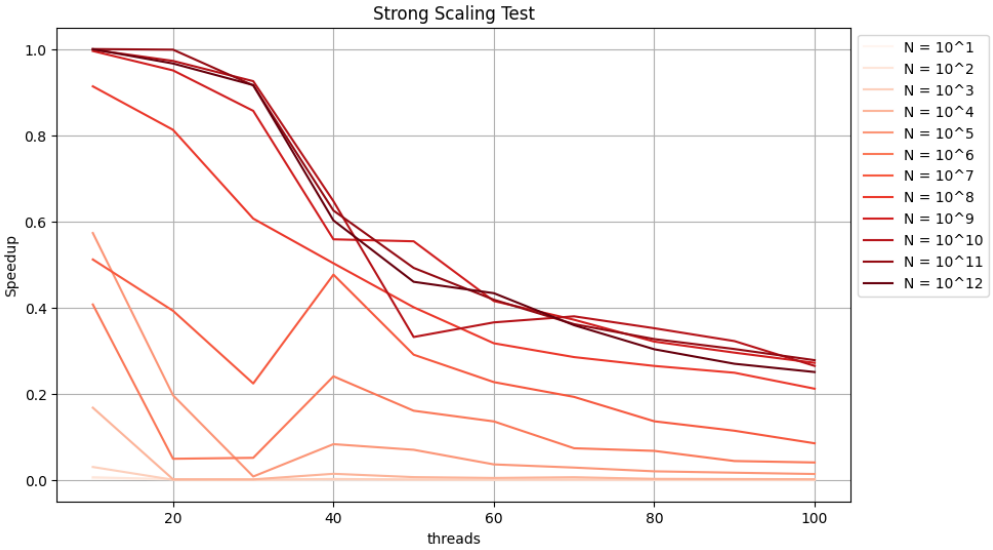


Figure 5: Strong scaling test

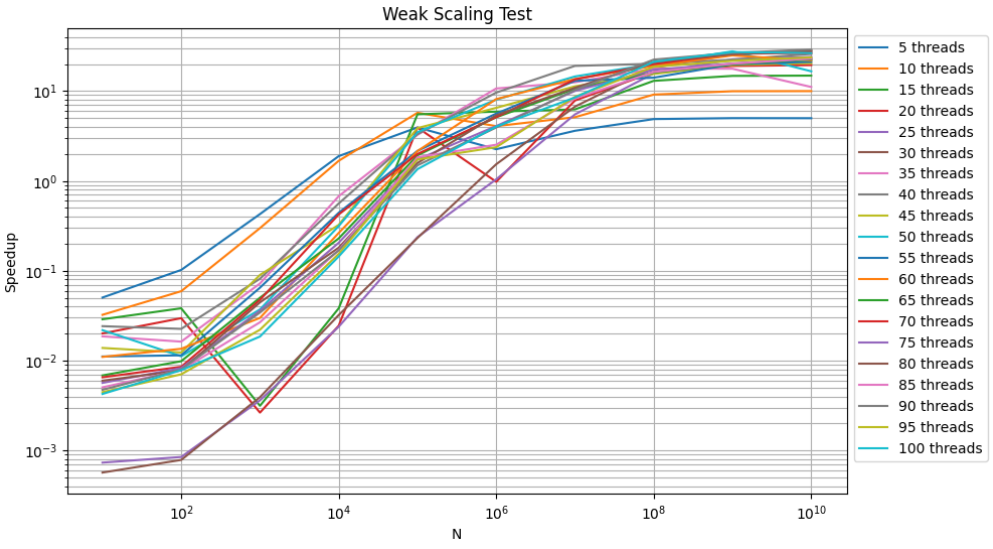


Figure 6: Weak scaling test