REPORT

Trapezoidal Rule using MPI

Subject	**Bigdata Processing (MS)**	**Professor**	**Jae-yeon-Park**
ID	**32204292**	**Major**	**Mobile System Eng.**
Name	**Min-hyuk-Cho**	**Submit Date**	**2024/11/01**

Index

1. Introduction

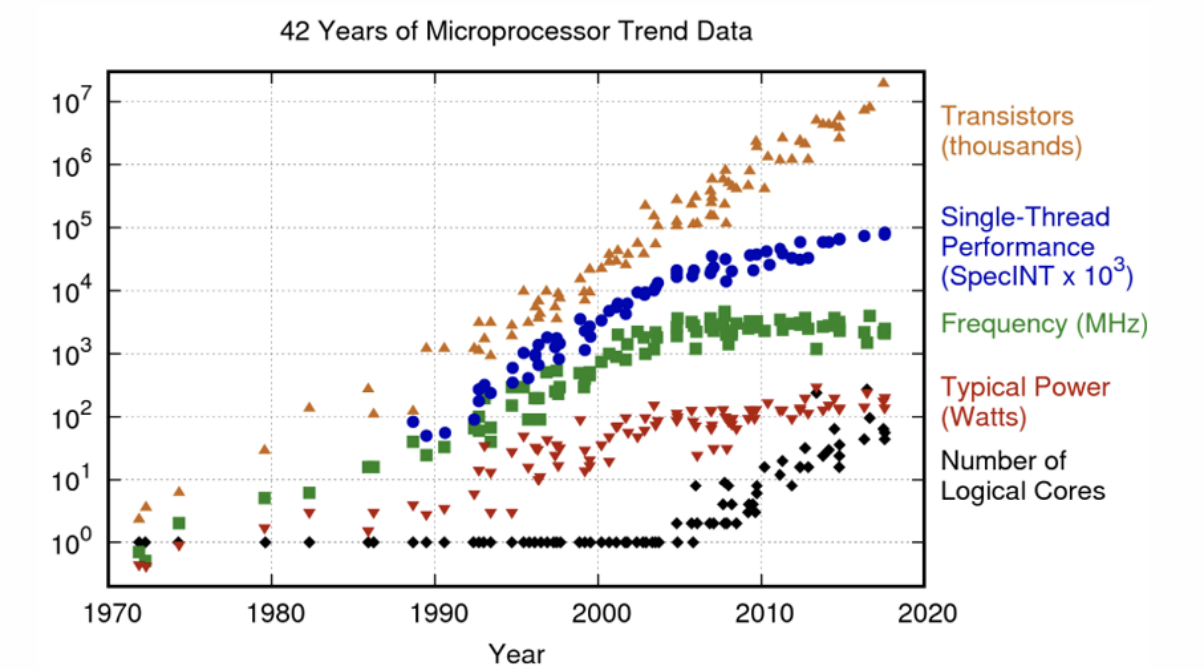

Fig 1. Why Parallel Programing?

Modern computer systems achieve substantial performance improvements in data processing through parallelization. In the past, single-core systems were predominant, and performance was enhanced by increasing the number of transistors, as illustrated in Fig 1. This approach was effective until the early 2000s; however, physical limitations such as thermal and power constraints began to impose limits on further performance gains through increased transistor counts. Consequently, systems transitioned from single-core to multi-core architectures, which not only improved computational performance but also proved effective for handling today's large-scale data processing requirements. Multi-core systems, employing multiprocessing and multithreading techniques, have optimized parallelism and enhanced performance, playing a critical role in the efficiency of computation-intensive tasks.

In this report, we will develop a parallelized program using MPI to implement the Trapezoidal Rule. Following program implementation, we will evaluate the performance of parallel programming through comparisons with a serial program, scalability testing, and precision testing. The structure of this report is as follows. In Chapter 2, we outline the requirements for parallel programming, and in Chapter 3, we introduce relevant concepts. Chapter 4 provides a detailed explanation of the implemented code, while Chapter 5 presents and analyzes the results, and finally, the report concludes with a summary of key findings.

2. Requirements

Table 1. Requirements Table

Index	Requirements
1	Don't change MPI_Wtime's location within the code.
2	Part 1 – Serial vs Parallel Execution Time Comparison • Setup – For both functions o $\boldsymbol{f(x)=x2}$ over the interval $[\mathbf{0},\mathbf{2}]$. o $\boldsymbol{f(x)=x4-3x2+x+4}$ over the interval [0, 2]. • Task Run the program with 4096 trapezoids in two configurations: 1. Serial execution. 2. Parallel execution with 4 processes using MPI. Measure the execution time for both configurations using MPI_Wtime()and compare the results. • Deliverables 1. Report the estimated integral for both functions in serial and parallel execution. 2. Include the execution time for each configuration. 3. Discuss the performance differences between the serial and parallel versions.
3	Part 2 – Scalability Test • Setup – Run the trapezoidal rule MPI program for both functions $\boldsymbol{f(x)=x2}$ over the interval $[\mathbf{0},\mathbf{2}]$. $\boldsymbol{f(x)=x4-3x2+x+4}$ over the interval [0, 2]. • Task 1. Increase the number of processes and trapezoids to observe how well your program scales. • Test Scenarios 1. Use 1, 2, 4, and 8 processes. 2. For each number of processes, test the trapezoidal rule with 256, 1024, 4096, and 16384 trapezoids. 3. Perform these tests for both $\boldsymbol{f(x) = x2}$ and $\boldsymbol{f(x) = x4 - 3x2 + x + 4}$. • Metrics 1. Measure the speedup 2. Measure the efficiency • Deliverables 1. Submit the measured speedup and efficiency for each function. 2. Include a table and graph showing speedup and efficiency as a function of the number of processes and trapezoids for the functions: $\boldsymbol{f(x) = x2}$. 3. Include a table and graph showing speedup and efficiency as a function of the number of processes and trapezoids for the functions: $\boldsymbol{f(x) = x4 - 3x2 + x + 4}$. 4. Discuss the performance of the results on both functions.
4	Part 3 – Precision Test • Setup – For both functions $\boldsymbol{f(x)=x2}$ over the interval $[\mathbf{0},\mathbf{2}]$.

	$f(x)=x4-3x2+x+4$ over the interval [0, 2]. • Task 1. Run the program with 2 processes and increase the number of trapezoids: 10, 40, 160, and 640. Compare the estimated integral with the exact integral values: For $f(x) =x2$, the exact value is 8 3 ≈2.6667. For $f(x) =x4-3x2+x+4$, the exact value is 42 5 =8.4. • Deliverables 1. Submit a precision report comparing your estimated results with the exact values for both functions. 2. Discuss how the number of trapezoids affects precision for both functions.

3. Concepts

This section covers the concepts essential for implementing the program.

3-1. Parallel Software

Parallel software utilizes multiple processing units, such as CPU cores and GPUs, to execute numerous tasks simultaneously. Modern computer systems are built with multiple cores, making efficient use of them a critical aspect of software design. The primary goal of parallel software is to reduce total execution time by distributing tasks effectively. To achieve this, it is necessary to determine an appropriate distribution method, typically Data Parallelism or Task Parallelism.

Data parallelism divides large data sets across multiple processors, enabling each processor to handle only a specific portion of the data set. In contrast, task parallelism assigns different tasks or functions to separate processors for execution. After data distribution, the processing of these tasks is handled by processes or threads operating in parallel.

A process is an instance of a running program with its own memory space, enabling parallel execution, especially in distributed memory systems, where interaction occurs through message passing. A thread, on the other hand, is a smaller unit derived from a single process and allows parallel execution by creating multiple threads within the same process. In multi-threaded systems, proper synchronization is essential to manage resource access and prevent conflicts. Synchronization tools like Mutex, Semaphore, and Barrier are commonly used for this purpose.

Another notable feature of parallel programming is Nondeterminism, meaning that the order of process and thread execution is not predetermined. While the output may be correct, the order of execution may vary with each program run due to scheduling algorithms. Consequently, the order of thread or process execution may differ on each run

3-2. Performance

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Fig 2. Table of Performance Parallel Program

After developing a parallel program, performance evaluation becomes essential. The primary goal of parallel programming is to maximize hardware resource utilization while minimizing execution time. As shown in Fig 2, **Speedup** and **Efficiency** may vary depending on the dataset. By analyzing these performance metrics, it is possible to identify the optimal dataset that maximizes hardware utilization and minimizes execution time.

- $S\ (Speedup) = \frac{T_{serial}}{T_{parallel}}$
- $E\ (Efficiency) = \frac{S\ (Speedup)}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{p \times T_{parallel}}$

Speedup is defined as the ratio between the execution times of the parallel program and its serial counterpart. A higher speedup indicates more effective parallelization. **Efficiency** is calculated by dividing speedup by the number of processors used, representing the overall utilization of system resources. This metric helps determine how effectively resources are used as the number of processors increases.

3-3. Parallel Program Design

Designing parallel programs involves structuring the program to effectively utilize multicore systems. Four key principles guide this process: **Partitioning**, **Communication**, **Aggregation**, and **Mapping**.

Firstly, **Partitioning** breaks down a large problem into smaller, independent tasks or data pieces that can be processed concurrently. Next, **Communication** design ensures that processors, while handling their assigned subtasks, can exchange data or synchronize as needed. This step establishes the means for inter-processor data transfer and interaction.

After task completion, **Aggregation** gathers the results from each processor to consolidate the final output. Finally, **Mapping** optimizes the program by applying data locality to specific processors and minimizing communication overhead, ultimately reducing execution time.

Following these four design principles, parallel programs can be developed and implemented using tools like MPI, as discussed in the subsequent sections.

3-4. MPI (Message Passing Interface)

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);  // Initialize the MPI environment

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // Get number of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  // Get the rank of the process

    printf("Hello from rank %d out of %d processors\n", world_rank, world_size);

    MPI_Finalize();  // Finalize the MPI environment
}
```

Fig 3. Skeleton code of MPI Program

MPI (Message Passing Interface) is a standardized message-passing system widely used across parallel computers, particularly in distributed memory systems. One of MPI's key features is **portability**, allowing it to function across various architectures. It also offers **scalability**, supporting efficient communication among numerous processors. MPI facilitates communication through **Processes**, **Ranks**, and **Communicators**. **Rank** is an integer identifier that allows specific messages to be sent to and received from designated processes. **Communicator** defines a group of processes that can communicate with each other.

Fig 3 illustrates the basic structure of an MPI program. The MPI_Init function initializes the MPI environment, while MPI_Comm_size registers the number of processes within MPI_COMM_WORLD. The MPI_Comm_rank function assigns a unique rank to each process, distinguishing them within the group.

In this report, the implemented program distributes tasks based on **Data parallelism** across multiple processes. The partial results calculated by each process are then collected and aggregated into a result using the MPI_Reduce function. As part of MPI's **Collective Communication** operations, MPI_Reduce is effective for both distributing and aggregating data. Specifically, MPI_Reduce performs an operation on the gathered results and stores the final output in a specified buffer.

4. Implements

This section describes the program being implemented and explains the corresponding code.

4-1. Trapezoidal Rule using MPI

$$\circ Integral \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right]$$

Fig 4. Formula of Trapezoidal Rule

The Trapezoidal Rule approximates the area under a curve by dividing the integration interval into trapezoids and calculating the area of each. Applying MPI to this method allows for a parallel implementation. The total interval [a, b] is divided into n−1n - 1n−1 subintervals, each assigned to a separate process for parallel computation.

Each process calculates the area of the trapezoid for its assigned interval, and these partial results are combined to produce the final integral result. This parallel approach significantly improves performance compared to a single process performing all calculations, as it reduces the overall execution time. Fig 4 presents the formula representing this method. Given the interval [a, b] divided into n−1n - 1n−1 subintervals, the area of each trapezoid is calculated and summed to approximate the integral.

This program leverages MPI's data parallelism by distributing subinterval computations across processes, then uses the MPI_Reduce function to aggregate the results from each process into the final result.

4-2. Build environment

Before explaining the implemented code, the development environment and setup used to run the program are as follows:

- ✔ Development Environment: Ubuntu 22.04, VSCode
- ✔ Programming Language: C
- ✔ Program Execution Steps:

STEP 1) mpicc -g -Wall -o <Execution File Name>

STEP 2) mpiexec -n <Process #> <Execution File Name>

4-3. Part 1 – Serial vs Parallel Execution Time Comparison

In this section, the Part1 program was implemented based on the program requirements outlined in Chapter 2.

```c
// Function to integrate
double f(double x) {
    return x * x; // f(x) = x^2
}

double f2(double x){
    return f(x) * f(x) - 3*f(x) + x + 4; // f(x) = x^4-3x^2+x+4
}

double Trap_2(double left_endpt, double right_endpt, int trap_count, double base_len) {
    double estimate_2, x;
    int i;

    estimate_2 = (f2(left_endpt) + f2(right_endpt)) / 2.0;
    for (i = 1; i <= trap_count - 1; i++) {
        x = left_endpt + i * base_len;
        estimate_2 += f2(x);
    }
    estimate_2 = estimate_2 * base_len;

    return estimate_2;
}

// Output the result and total time
if (my_rank == 0) {
    printf("========================================================================================================\n\n");
    printf("Part 1: Serial vs Parallel Execution Time Comparasion\n");
    printf("%d Processes used\n", comm_sz);
    printf("With n = %d trapezoids, our estimate of the integral from %f to %f = %.15e\n", n, a, b, total_int);
    printf("Total elapsed time = %f seconds\n\n", elapsed);
    printf("========================================================================================================\n\n");
}
```

Fig 5. Part1 Source Code

Fig 5 shows a portion of the Part1 source code. Initially, a function was created to set up the test environment, and the f2 function was implemented by reusing the original f function. Based on the provided code, the Trap_2 function was developed. This function tests f2 by calculating the trapezoidal area for the segments assigned to each process. This setup allows us to compare the execution time between the parallelized and serial versions of the program.

4-4. Part 2 – Scalability Test

```
for(int i = 0; i < 4; i++){
    // Start timing the entire program
    start[i] = MPI_Wtime();

    h[i] = (b - a) / n[i]; // width of each trapezoid
    local_n[i] = n[i] / comm_sz; // number of trapezoids for each process

    // Each process calculates its local interval
    local_a[i] = a + my_rank * local_n[i] * h[i];
    local_b[i] = local_a[i] + local_n[i] * h[i];

    // Calculate the local integral

    local_int[i] = Trap(local_a[i], local_b[i], local_n[i], h[i]);
    // local_int[i] = Trap_2(local_a[i], local_b[i], local_n[i], h[i]);

    // Sum up the integrals from all processes using MPI_Reduce
    MPI_Reduce(&local_int[i], &total_int[i], 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // End timing the entire program
    finish[i] = MPI_Wtime();
}
```

Fig 6. Part2 Source Code

```
#!/bin/bash

for n in 1 2 4 8
do
    echo "Running with $n processes:"
    if [ $n -eq 8 ]; then
        mpiexec --oversubscribe -n $n MPI_Trapezoid_Part2_f2
    else
        mpiexec -n $n MPI_Trapezoid_Part2_f2
    fi
    echo "----------------------------------------"
done
```

Fig 7. Part2 Bash Script

In this section, the scalability of the Part2 program is evaluated. Fig 6 shows a portion of the Part2 source code, where various numbers of processes and trapezoid counts were set for scalability testing across multiple scenarios. Arrays were used to implement variables, allowing flexible testing under different configurations.

Fig 7 presents a Bash script created to automate the execution of the program with different

numbers of processes. This script enables the program to run across various process configurations. Since the test environment does not support 8 processes, the --oversubscribe option was added to enable the execution of tests in an 8-process environment.

4-5. Part 3 – Precision Test

```
int n[4] = {10, 40, 160, 640}; // Array of number of Trapezoid
```

Fig 8. Part3 Source Code

In Part 3, the precision of the program is evaluated. Since there are no major differences from the Part 2 source code, only the modified sections are shown in Fig 8. This test uses 2 processes and 4 trapezoid counts (10, 40, 160, 640) to evaluate accuracy, with these trapezoid values arranged in an array for easier configuration.

Additionally, as this experiment is conducted in a fixed environment with 2 processes, no additional script was needed. This setup allows us to compare precision across the specified trapezoid counts and assess the impact of parallel processing on accuracy.

5. Results

For $f(x) = x^2$, the exact value is $\frac{8}{3} \approx 2.6667$.
For $f(x) = x^4 - 3x^2 + x + 4$, the exact value is $\frac{42}{5} = 8.4$.

Fig 9. Value of Functions

This section presents the results of the tests performed using the implemented source code. Fig 9 displays the integral values of the target functions prior to evaluating the results.

5-1. Part 1 – Serial vs Parallel Execution Time Comparison

```
Part 1: Serial vs Parallel Execution Time Comparasion
4 Processes used
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000219 seconds
```

```
Part 1: Serial vs Parallel Execution Time Comparasion
4 Processes used
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364245e+00
Total elapsed time = 0.000319 seconds
```

Fig 10. Output of Part 1 Program

Table 2. Result of Part 1 Program

	Serial Execution	Parallel Execution
Number of Process	1	4
Area	2.6666	8.4000
Execution Time (sec)	0.000219	0.000319

Fig 10 shows the results for Part 1, and Table 2 summarizes these findings. Analyzing the results, we see that both Serial Execution and Parallel Execution produce values close to the theoretical integral values shown in Fig 9. However, contrary to the expectation that parallel execution would result in a faster runtime, serial execution proved quicker. This discrepancy is likely due to the overhead involved in creating and terminating processes in parallel execution, which added additional time to the overall execution.

5-2. Part 2 – Scalability Test

Table 3. Result of Part 2 Program

p / Trapezoids			1	2	4	8
256	f1	Area	2.66668	2.66668	2.66668	2.66668
		Execution Time	0.000002	0.00002	0.00006	0.00062
		S	1	0.1	0.03	0.003
		E	1	0.05	0.0075	0.000375
	f2	Area	8.4001	8.4001	8.4001	8.4001
		Execution Time	0.000007	0.000395	0.000146	0.000386
		S	1	0.01	0.04	0.018
		E	1	0.005	0.01	0.002
1024	f1	Area	2.66666	2.66666	2.66666	2.66666
		Execution Time	0.000003	0.000002	0.000002	0.000004
		S	1	1.5	1.5	0.75
		E	1	0.75	0.375	0.09
	f2	Area	8.400006	8.4000	8.4000	8.4000
		Execution Time	0.000016	0.00001	0.00001	0.000006
		S	1	1.6	1.6	2.6
		E	1	0.8	0.4	0.325
4096	f1	Area	2.66666	2.66666	2.66666	2.66666
		Execution	0.000010	0.000006	0.000006	0.000003

		Time				
		S	1	1.6	1.6	3.3
		E	1	0.8	0.4	0.4
	f2	**Area**	8.400000	8.400000	8.400000	8.4000000
		Execution Time	0.0001	0.00004	0.00004	0.00001
		S	1	2.5	2.5	10
		E	1	1.25	0.6125	1.25
16384	**f1**	**Area**	2.66666	2.66666	2.66666	2.66666
		Execution Time	0.000041	0.000029	0.000012	0.000017
		S	1	1.4	3.4	2.4
		E	1	0.7	0.85	0.3
	f2	**Area**	8.400000	8.400000	8.400000	8.3951089
		Execution Time	0.000468	0.000128	0.000126	0.000095
		S	1	3.6	3.7	4.9
		E	1	1.8	0.925	0.6125

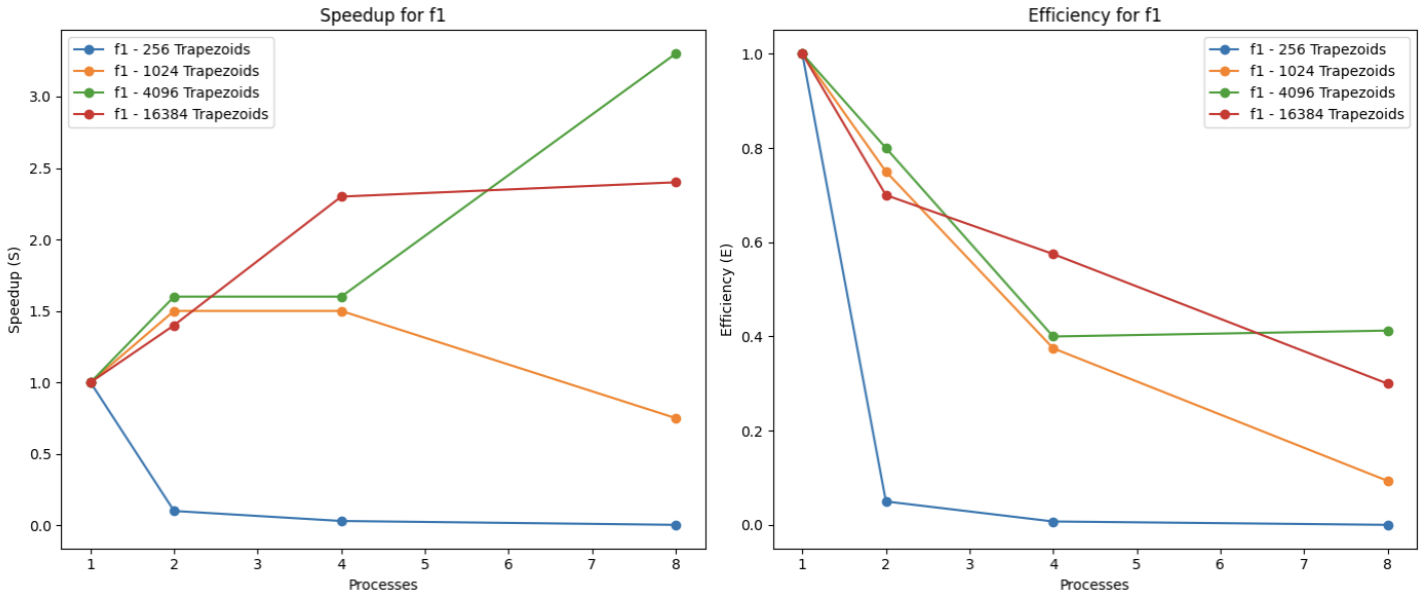

Fig 11. Graph of Part 2's f1

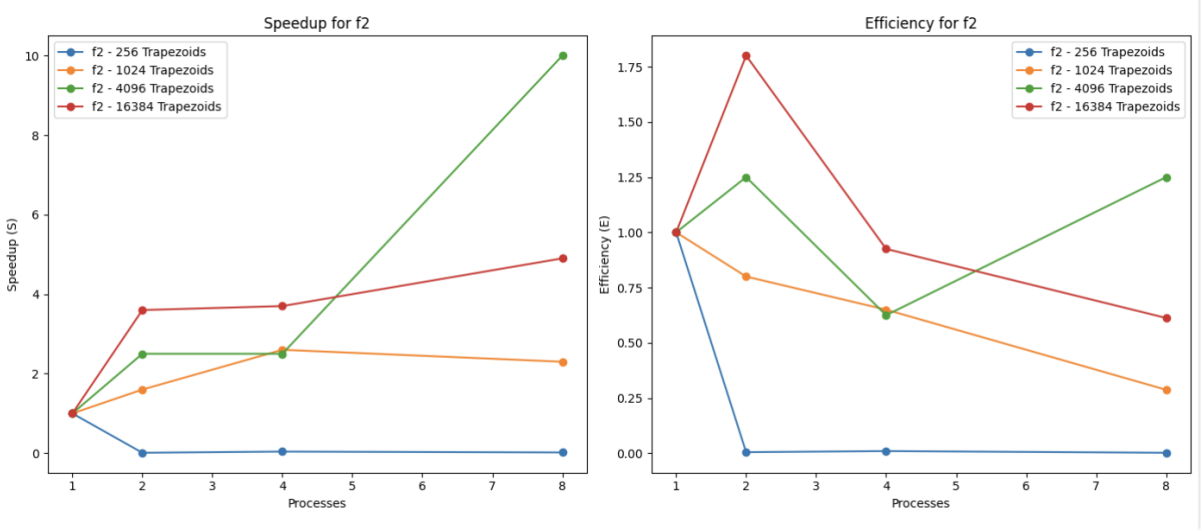

Fig 12. Graph of Part 2's f2

Table 2 summarizes the execution results for the f1 and f2 functions in Part 2 program. These results are visually represented in Fig 11 and Fig 12.

Starting with Fig 11, we observe that in the f1 function, when the number of trapezoids is minimal, increasing the number of processes led to a decrease in speedup. However, from 4096 trapezoids onward, speedup improved steadily as the number of processes increased. When the number of trapezoids was too large (e.g., 16384), further speedup improvements beyond 4096 trapezoids were minimal. Examining the Efficiency of f1, we see that the lowest efficiency was recorded with the smallest number of trapezoids, and even with an increased number of trapezoids, the efficiency did not exceed 1. Nonetheless, the best efficiency was achieved with 4096 trapezoids, where the speedup was also highest.

Analyzing Fig 12, we find similar trends for the f2 function. The highest speedup occurred with 4096 trapezoids, while speedup declined when the trapezoid count was minimal. In terms of efficiency, the f2 function exhibited better efficiency than the f1 function, showing stable efficiency with 4096 trapezoids. Conversely, the lowest efficiency was recorded when the number of trapezoids was minimal.

In summary, 4096 trapezoids appear to be the most suitable dataset size for parallel programming, as too few data points negatively impact performance.

5-3. Part 3 – Precision Test

Table 4. Result of Part 3 Program

p=2 \ Trapezoids	10	40	160	640
f1	2.68	2.6675	2.6667	2.6666
f2	8.4665	8.4041	8.4002	8.4000

Table 4 presents the results for the f1 and f2 functions based on the number of trapezoids when using 2 processes. According to Fig 9, the target result for f1 is 2.6667 (or exactly 8/3), and for f2, it is 8.4. Analyzing Table 4 reveals that the f1 function achieves the most accurate result with 160 trapezoids, while the f2 function reaches its highest accuracy with 640 trapezoids.

Therefore, 160 trapezoids for f1 and 640 trapezoids for f2 provide the most accurate results. However, considering that the exact area for f1 is 8/3, it becomes evident that with 640 trapezoids, both f1 and f2 yield results closest to the true values. This suggests that 640 trapezoids provide the most precise dataset for accurate calculations in both functions.

6. Conclusion

In this report, we implemented parallel software using the Trapezoidal Rule with MPI. First, we defined the requirements for implementing the program in Chapter 2 and outlined relevant concepts to establish a foundation in parallel programming. Subsequently, we examined the ideas and code used in the implementation, and in Chapter 5, we analyzed the results through tables and graphs to compare serial and parallel execution.

Through the Scalability Test, we identified the number of trapezoids that most significantly impacted the speedup and efficiency of the parallel program. Additionally, the Precision Test allowed us to determine the optimal number of trapezoids to ensure accurate results. This analysis revealed that an increase in dataset size does not always guarantee speedup, and there may be a margin of error depending on the dataset size.

This project provided an in-depth understanding of parallel programming, offering both theoretical insights and valuable experience in code implementation. Ultimately, I gained confidence in my ability to implement more complex parallel programs and to conduct performance evaluation and analysis.