Trapezoidal Rule for Estimating Integrals With MPI

Michael Dandrea

***Abstract*—This report presents the design and implementation of a parallelized trapezoidal integration program using the Message Passing Interface (MPI). The trapezoidal rule is a numerical method used to approximate the definite integral of a function. To improve computational efficiency, particularly for large-scale problems, the integration process is distributed across multiple processors. By dividing the total number of trapezoids among the processes, each processor computes a portion of the integral, and the results are aggregated to form the final solution. The implementation supports a flexible number of processes and intervals, leveraging Quinn's MPI macros for load balancing. A series of experiments were conducted to assess the program's performance and scalability, focusing on the execution time, speedup, and efficiency for different problem sizes and processor counts. The results demonstrate the effectiveness of the parallel approach, particularly for large problem sizes, where significant speedup and computational gains are observed.**

I. Trapezoidal Rule for Estimating Integrals

The trapezoidal rule is a numerical method for estimating the definite integral of a function. It works by approximating the region under a curve as a series of trapezoids, rather than using rectangles as in the Riemann sum. The formula for the Trapezoidal Rule is as follows:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2n}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right]$$

where:

- a and b are the limits of integration,
- n is the number of subdivisions (trapezoids),
- and $f(x_i)$ is the value of the function at the point x_i.

The method improves accuracy by averaging the function's values at the endpoints of each subdivision, forming trapezoids, whose areas are then summed. It is particularly useful when the function is difficult to integrate analytically or when an approximate solution is sufficient. The accuracy of the result increases as the number of subdivisions, n, increases.

A. How Is This Different From A Riemann Sum

A Riemann sum is a method for estimating the value of a definite integral. It involves dividing the region under a curve into several small subdivisions, then approximating the area of each subdivision using rectangles. The total area is found by summing the areas of these rectangles. For each rectangle in a Riemann Sum:

- The width is a small change in the x-axis (Δx)
- The height is determined by the function value at a specific point within each subdivision (left, right, or midpoint).

The formula for a Riemann Sum is as follows:

$$S_n = \sum_{i=1}^{n} f(x_i^*)\Delta x$$

where:

- a and b are the limits of integration,
- x_i is a point in each subdivision,
- and n is the number of subdivisions.

II. Design and Implementation

The design of the program revolves around parallelizing the trapezoidal rule for numerical integration using MPI (Message Passing Interface). By dividing the interval of integration across multiple processes, each process calculates its portion of the integral independently. The results are then combined to produce the final approximation of the integral.

A. Program Structure

This program was split into three main files:

- mpi_trap.c: This is the main driver of the program and handles MPI initialization, work distribution, and communication between processes. It also includes timing operations to measure performance.
- trap.c: This file contains all functions called by mpi_trap.c that are not part of an included library. This includes the implementation of the trapezoidal rule and command line parsing.
- trap.h: This header file contains function definitions for trap.c and utility macros from Quinn's header file for use in the program.

The program makes use of a Makefile to do separate compilational units for both of the .c files.

B. Command Line Parsing

This program is designed to accept command line arguments to specify the limits of integration, number of trapezoids, and the function to integrate. The arguments are as follows:

- -a: Number to integrate from,
- -b: Number to integrate to,
- -n: Number of trapezoids,
- -f: Function to integrate.

All of these are required arguments, and the program returns a usage statement if any are missing. The functions to integrate are predefined as follows:

- 1: $f(x) = x$
- 2: $f(x) = x^2$
- 3: $f(x) = sin(x)$

Having three predefined functions to integrate was a requirement of the project, making it easier to focus on the MPI implementation of the trapezoidal rule. The following code snippet on the next page shows how the command line parsing is handled:

```
while ((opt = getopt(argc, argv,
"n:a:b:f:")) != -1) {
    switch (opt) {
        case 'n':
            *n_p = atoll(optarg);
            break;
        case 'a':
            *a_p = atof(optarg);
            break;
        case 'b':
            *b_p = atof(optarg);
            break;
        case 'f':
            *f_p = atoi(optarg);
            break;
    }
}
```

The values for a, b, n, and f are collected by the root process (rank 0) and then distributed to all other processes using MPI's MPI_Bcast() function. This ensures all processes work with the same input data, minimizing the chances of inconsistency.

C. Parallel Integration

The trapezoidal rule was implemented to distribute the computational workload among multiple processors using MPI. The idea is to divide the integration interval into subintervals, which are assigned to each process for individual computation. In the program, the interval is divided by the total number of trapezoids, n, and each process is assigned a portion of the interval based on its rank. This is done by using MPI functions to determine the number of processes and each process's rank. For instance, the rank of the process is determined by the following code:

```
MPI_Comm_rank(MPI_COMM_WORLD,
&my_rank);
```

Each process determines its local bounds using the following:

```
local_a = a + BLOCK_LOW(my_rank,
comm_sz, n) * h;

local_b = local_a + local_n * h;
```

where BLOCK_LOW is defined in the Quinn code that was provided for the project. From there, each process then calls the Trap() function, which takes the left endpoint, right endpoint, number of trapezoids, base length and function as parameters. It then executes the following for loop in the Trap() function to estimate its portion of the estimate:

```
estimate = (func_ptr(left_endpt)
+ func_ptr(right_endpt)) / 2.0;
for (i = 1; i < trap_count; i++) {
    x = left_endpt + i * base_len;
    estimate += func_ptr(x);
}
estimate = estimate * base_len;
```

Once the programs have computed their portions of the sum, MPI_Reduce() is called to do a reduction sum of the processes values, which is the final estimate for the integral.

III. Running on Expanse

This project was run on the Expanse Supercomputer at the San Diego Supercomputing Center (SDSC). Expanse is an excellent platform for running large-scale parallel programs using MPI. This section outlines the steps required to compile, run, and analyze the results of the MPI-based trapezoidal integration program on Expanse.

The first step was to upload the project to Expanse. This was easy as Expanse allows you to do that through their website. After that, a bash script was made to run the project on Expanse and safe the results to a file for use later. The script submits the the job to the compute node on Expanse. The program was run a total of 80 times with various numbers of processors and trapezoids, with np ranging from 1 to 16, and n starting at 100 million and ending at 500 million, incrementing by 100 million. Finally, the restults of these executions were written to a text file to be interpreted.

IV. Results

In this section, we present the results obtained from running the MPI trapezoidal integration program on various values of N (number of trapezoids) and P (number of processors). The goal was to assess the performance, scalability, and efficiency of the parallel implementation across different problem sizes. A Python script was written to take the results from running on Expanse and display that data as graphs, along with the ideal line.

A. Timing Results

The program was executed for values of N ranging from 10,000 to 1.5 billion trapezoids, and for each value of N, the number of processors P was varied from 1 to 16. The wall-clock execution time was recorded for each combination of N and P. Figure

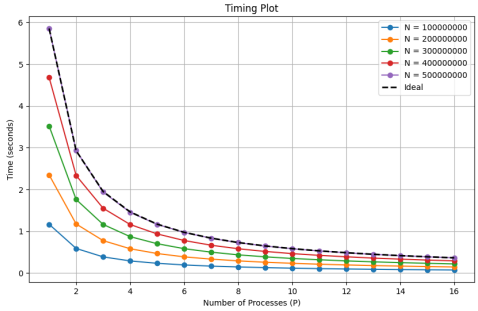

Figure 1: Timing Graph

1 shows the timing results for five values of N as the number of processors increases, as well as the ideal line.

$$Time(seconds) = f(N, P)$$

As expected, for each value of N, the execution time decreases as the number of processors increases. However, the rate of reduction diminishes as P increases, due to communication overhead becoming a more significant factor as the number of processors increases. For larger values of N, the timing curve flattens, indicating better utilization of the available processors. Something interesting about this graph is that the speedup for n = 500 million is equal to the ideal speedup. This could be because the time to run these processes is so small (about 10 seconds).

B. Speedup Analysis

Speedup is defined as the ratio of the execution time with one processor (T1) to the execution time with P processors (Tp):

$$Speedup = T_1/T_p$$

Ideally, the speedup should be linear, meaning that doubling the number of processors should halve the execution time. Figure 2 shows the speedup curves for five values of N, as well as the ideal.

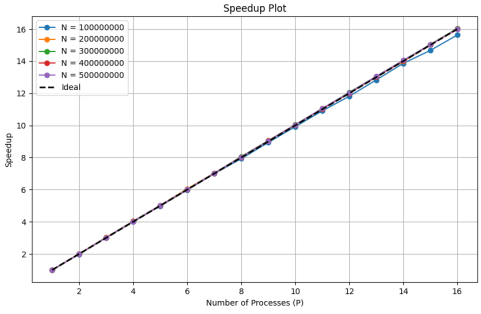

Figure 2: Speedup Graph

The lines for almost all of the executions are exactly ideal. This could be for two reasons, or maybe both of them. Like mentioned before, he amount of time to run these processes is so short (T = 10 seconds on one processor) that it is skewing the results. The other reason this could happen is because most, if not all of the work being parallelized is additions of long long ints, which is already extremely fast. In addition, there is not much communication overhead required from MPI, so the speedup is very close to if not ideal.

C. Efficiency Analysis

Efficiency is a measure of how well the processors are utilized and is calculated as:

$$Efficiency = Speedup/P$$

In an ideal scenario, efficiency would remain constant as the number of processors increases. Figure 3 shows the efficiency curves for five values of N as well as the ideal line (100%). For smaller problem sizes, efficiency drops off significantly as P increases, indicating that communication overhead and idle time increase with more processors. However, for larger values of N, the efficiency remains relatively high, even over 100%, even as the number of processors increases, showing that the program scales well for large problem sizes.

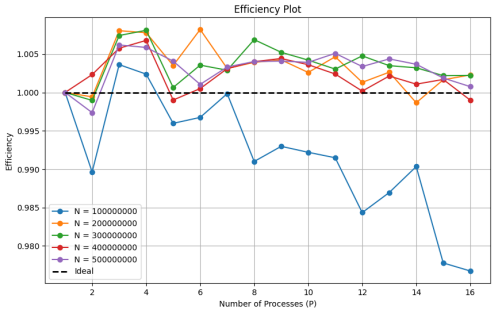

Figure 3: Efficiency Graph

The reason for efficiency being over 100% is once again because of how short the execution times were for these programs. The theories for Efficiency, Speedup and Timing are for much larger problem sizes, so sometimes with smaller sample sizes the results can look a bit off.