Homework 02 MPI Numerical Integration

Ryan McCormick
Department of Computing Sciences
Coastal Carolina University
Conway, SC, USA
rlmccormi@coastal.edu

***Abstract*—Evaluating speedup and efficiency in parallel computing systems is valuable in order to understand the effects of compute resources on a given problem set. Analyzing this relationship can help with better understanding how problem size and number of compute resources are linked. This is valuable when trying to assess how many resources are needed for a given task, especially in a time like today where computational resources are in such high demand. The data that comes from analyzing this relationship can show the point at which return on investment is diminished (the point where efficiency is lower than what is being accomplished). Throughout the report that follows, there will be a walk-through of the problem that was being tested, an explanation of how these tests were conducted and a presentation of the results. This will include a breakdown of the formulas used and how the data was gathered and processed to create the graphs featured in this paper.**

***Index Terms*—efficiency, speedup, performance, MPI**

I. Introduction

Parallel systems, when used with code that is properly parallelized, can take a problem that would normally take a long time to run serially and accomplish the same task much faster. Some problems lend themselves very well to being parallelized, while others do not. When a problem can be parallelized, this can lead to huge reductions in compute time.

One problem that lends itself to parallelization is calculating the area under a curve using the trapezoidal rule. Riemann Sums is the method of approximating the area below a curve by sub-dividing the area into smaller, simpler shapes and then adding the area of each rectangle to approximate the total area under the curve.

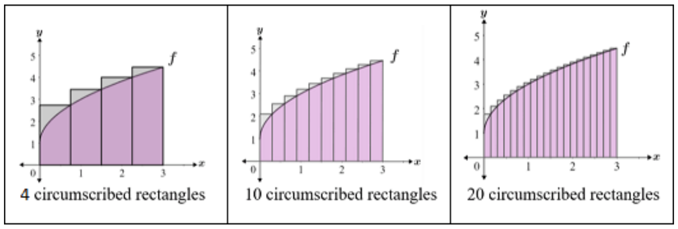


Fig. 1. Riemann Sums Example [1]

Figure 1 shows how the area can be divided into any number of rectangles, and the accuracy of the approximation is improved by increasing the number of rectangles. This formula is a great candidate for parallelization because you can break each calculation for the area of one rectangle as its own process that can be passed to a processor. This area can then be added in a reduction sum to find the total area under the curve. This is the formula for the Riemann Sum approximation:

$$\sum_{i=1}^{n} f(x_i)\Delta x_i \tag{1}$$

This technique of approximating the area under the curve is what was used to test the speedup and efficiency of parallel computing. The formula that was selected to give the program an appropriate amount of work was as follows:

$$f(x) = sin(x) \tag{2}$$

The graph for this equation is seen in the following figure.

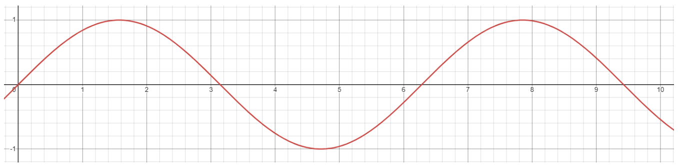


Fig. 2. Sin Curve Example

The more accurate way to estimate this area is though the use of the trapezoidal rule however. This uses trapezoids instead of rectangles, giving a much more accurate approximation for the area under the function. Figure 3 is an example of how the trapezoid rule works on the function $f(x) = 3ln(x)$ from $x = 2$ to $x = 8$

The trapezoid rule is the following:

$$h(\frac{b_1 + b_2}{2}) \tag{3}$$

[2]

h	: Height of the function at the location of the trapezoid.
b_1	: Lower bound of the trapezoid.
b_2	: Upper bound of the trapezoid.

This area of a trapezoid was then calculated for each subdivision and then added together like the Riemann sums

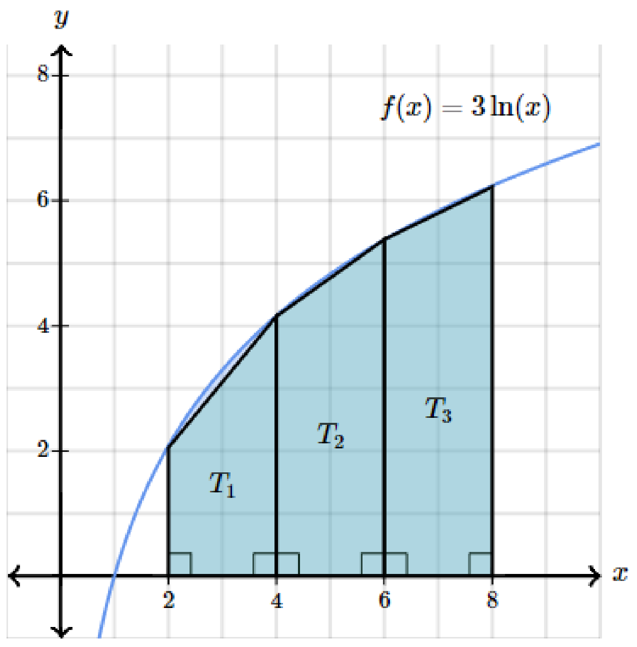


Fig. 3. Trapezoid Rule Example [2]

mentioned before. When gathering data for this research, the area under this curve was calculated from $x = 0$ to $x = 1000$. This calculation was then performed with different counts for the number of trapezoids, and the program was given different numbers of processors to spread the work between.

This paper will continue into greater detail about how the rest of this research was conducted and what process was followed. This paper is organized as follows. Procedure II covered the process and theory behind the experimentation, while Section III provides an outline for the software that was used. In Section IV, the outcomes of the research are shown and discussed. Section V concludes the paper.

II. Procedure

The idea of this project was to parallelize the sums of the trapezoid rule and approximate the area under the $f(x) = sin(x)$, while changing the number of subdivisions along with the number of processors given to help with the approximation. These calculations were timed and that data is what was analyzed see what differences occurred under the different circumstances.

The metrics that were chosen to evaluate these tests were as follows:

- Runtime of parallelized code
- Speedup
- Efficiency

Each time that the program was executed, three values were recorded: runtime (as mentioned before), number of processors, and number of subdivisions used in the calculation. These values then allowed for a few equations to be applied so that the above metrics could be calculated. The way that Speedup is calculated is the following equation:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{4}$$

[3]

T_{serial}	: Time to run the program with one processor
T_{parallel}	: Time to run the program in parallel

The Efficiency is something that was then calculated based on Speedup value. Efficiency is found given the following formula:

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p * T_{\text{parallel}}} \tag{5}$$

[3]

T_{serial}	: Time to run the program with one processor
T_{parallel}	: Time to run the program in parallel
p	: The number of processors

The values for these metrics were then calculated for each iteration of the program with the different values for p and the different number of divisions being used for the approximation. This data was written to a CSV (comma separated values) by a bash script that was running each of the combinations.

```
DIVISIONS="150000000 250000000 350000000
    ↪ 450000000 550000000"
for p in {1..16}; do
    for n in $DIVISIONS; do
        echo "P= $p, N=$n"
        mpirun -np $p --oversubscribe
            ↪ ./mpi_trap -n $n -a 0 -b
            ↪ 1000 -f 3 > temp.txt
        t=$(grep "T:" temp.txt | awk
            ↪ '{print $2}')
        echo "$p, $n, $t" >> $OUTPUT_FILE
    done
done
```

This is the `for()` loop that was used in the bash script to run the program the necessary number of iterations to generate the data needed for the project. The `DIVISIONS` were chosen such that the largest value, when run on one processor would take 10 second to execute. This number was then decreased in even increments to get five different values that could be tested. The outer loop is set to iterate from 1 to 16 cores, while each time the inner loop is testing each of the subdivisions for the approximation. This information is passed into the command line arguments that execute the program and the results are written, with a header, to the output file. The data produced was then processed with a Python script to compare and graph the data (shown in the Results section below). The data was then processed using the equations discussed above, and then placed into graphs to show trends in the numbers. The relationships between the data were analyzed as follows:

- Time vs Process Count
- Speedup vs Process Count
- Efficiency vs Process Count

These relationships allow for the analysis of the point at which the return on investment is diminished in terms of problem size, number of processors used and differences in runtime. This type of data can be crucial in determining when more or less time, money and effort should be put into parallelization, or which area needs the most focus, the algorithm being used, the system that is running the program or the way that the parallelization is implemented.

To get the final data that was used in all of the calculations of this project, the program was executed on a compute node of the Expanse system. Expanse is a dedicated Advanced Cyberinfrastructure Coordination Ecosystem: Services and Support (ACCESS) cluster. This system belongs to the San Diego Supercomputer Center at the University of California San Diego and was designed with members of their team and Dell. The compute node that was used in testing had the following specifications:

- CPU Type - AMD EPYC 7742
- Nodes - 728
- Sockets - 2
- Cores/socket - 64
- Clock speed - 2.25 GHz
- Flop speed - 4608 GFlop/s
- Memory capacity - 256 GB DDR4 DRAM
- Local Storage - 1TB Intel P4510 NVMe PCIe SSD
- Max CPU Memory bandwidth - 409.5 GB/s

[4]

Running the program on this system made it possible to test the parallel nature of the program in an environment that is not replaceable on a personal machine. Testing from 1-16 processes on a high quality and isolated node allowed for higher precision than would be possible on a normal computer.

III. Code

This project was started with a program from the book An Introduction to Parallel Programming by Peter S. Pacheco from the University of San Francisco.

```c
double Trap(
      int f_choice,
      double left_endpt,
      double right_endpt,
      int    trap_count,
      double base_len) {
   double estimate, x;
   int i;

   estimate = (f(f_choice, left_endpt) +
       ↪ f(f_choice, right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(f_choice, x);
   }
   estimate = estimate*base_len;

   return estimate;
}
```

Listing 1. Trap

[3]

This section of code is designed so that it receives the upper and lower bounds for the section that is being approximated. It then passes each of these to the function [$f(x) = sin(x)$] that is being approximated. The return value from the function is then summed so that the area under the curve between the upper and lower bounds is found.

The way that this is beneficial in a parallel program is in the way that $Trap()$ is utilized in $main()$. Main is set up to take in the user's command line arguments and do all of the error handling, but the next thing that $main()$ does is what make this program efficient. Using the Message Passing Interface (MPI) allows for information to be communicated between separate processors. This allows for the spread-loading of the work needed to solve a given problem, and in this case, the problem was the approximation of the area under a given function, as previously discussed.

```c
1 h = (b-a)/n;
2 local_n = BLOCK_SIZE(my_rank, comm_sz,
      ↪ n);
3
4 local_a = a + BLOCK_LOW(my_rank,
      ↪ comm_sz, n) * h;
5 local_b = local_a + local_n*h;
6 local_int = Trap(f_choice, local_a,
      ↪ local_b, local_n, h);
```

Listing 2. Snippet from $main()$

[3] [5]

This code snippet generates separate sections of the overall approximation and allows for each section of the calculation to be spread between different processors. In Listing 2, line 1 is responsible for determining the width of each trapezoid based on the total range and the number of desired subdivisions. Lines 2 and 3 then use two macros that were written by Michael J. Quinn. These macros allow for the calculation of how the work should be divided so that work is shared evenly between processes in a parallel environment. They help to determine which portion of the work will be passed to each process.

```
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)
    ↪ (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) \
                     (BLOCK_HIGH(id,p,n) -
                         ↪ BLOCK_LOW(id,p,n)+1)
```

Listing 3. Quinn Macros

[5]

This technique greatly enhances the speed of the computation. By dividing the load between separate processes, they are each able to compute a local sum for the area of their trapezoids. Each of these sums is then passed back to the root process, and the global sum can be computed, which is the approximation for the area under the curve. This is the code that allows for all of the processes to pass back their sums to compute the global sum.

```
MPI_Reduce(&local_int, &total_int, 1,
    ↪ MPI_DOUBLE, MPI_SUM, 0,
    ↪ MPI_COMM_WORLD);
```

Listing 4. Sum Reduction

[3]

While all of the above code was executing there was timing data being collected on the parallel portion of the work. This was done through the built-in timing function from the MPI library.

```
start_time = MPI_Wtime();

/*
dividing the work between the proceses
finding the sums
adding the sums together
*/

end_time = MPI_Wtime();

elapsed_time = end_time - start_time;
```

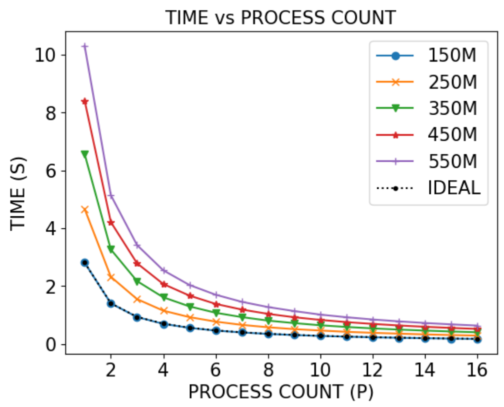
Listing 5. Timing

The values returned from the timing functions were then used to calculate the total time for the parallel portion of the code. This data was written to a file and used to conduct the analysis of the program.

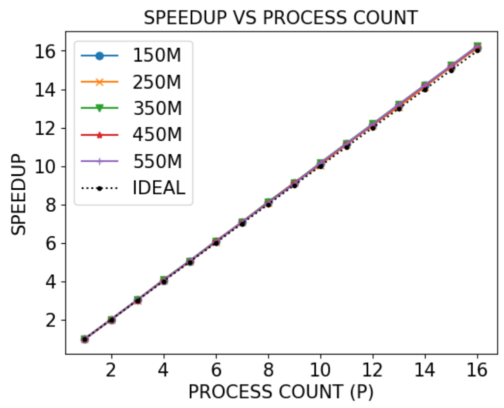
IV. Results

After testing on a local machine and testing on the Expanse node, as mentioned in the Procedures section, the data was then processed using a Python script, and these were the results.

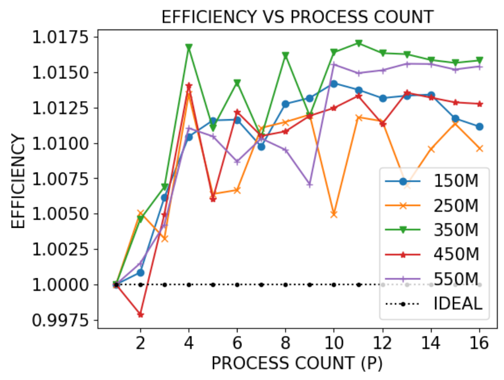
Time vs Process Count compared how the number of processors available to the program impacted the runtime of the parallel portion of the program. The time in this graph is in seconds.



Speedup vs Process Count compared how the program got faster as the number of processes was increased and how the problem size affected this change.



Efficiency vs Process Count compared how the number of processors and the problem size are related. This graph seems like there is a lot of variability, but remember the range of the y-axis is from 0.9975 to 1.0175. The efficiency should be below the ideal line, but in the testing data it appeared above. This was also replicated by a few other people that have run similar tests. We are unsure why this occurred.



V. Conclusion

This project presented a great opportunity to test how parallel programs can be affected by compute resources and how the problem size can also affect how well something benefits from parallelization. Using Expanse and getting to run the program in an actual testing environment was also a great experience. The results of this project were somewhat strange with the speedup being above the ideal case and the variance in the efficiency graph, but overall the data was similar to what was expected based on the theory behind the assignment. With more testing and averaging of the test results, there is a high possibility that the data would appear cleaner on the graphs. This is something that should definitely be tested in the future.

References

[1] Riemann sum. [Online]. Available: https://www.math.net/riemann-sum

[2] Trapezoid rule. [Online]. Available: https://www.khanacademy.org/math/ap-calculus-ab/ab-integration-new/ab-6-2/a/understanding-the-trapezoid-rule

[3] P. S. Pacheco, *An Introduction to Parallel Programming*, 2nd ed. 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States: Morgan Kaufmann Publishers, 2022.

[4] Expanse user guide. [Online]. Available: https://www.sdsc.edu/support/user$_guides/expanse.htmltech_summary$

[5] M. J. Quinn, *Parallel Programming in C with MPI and OpenMPI*, 1st ed. 1221 Avenue of the Americas, New York, NY 10020, United States: McGraw-Hill, 2004.