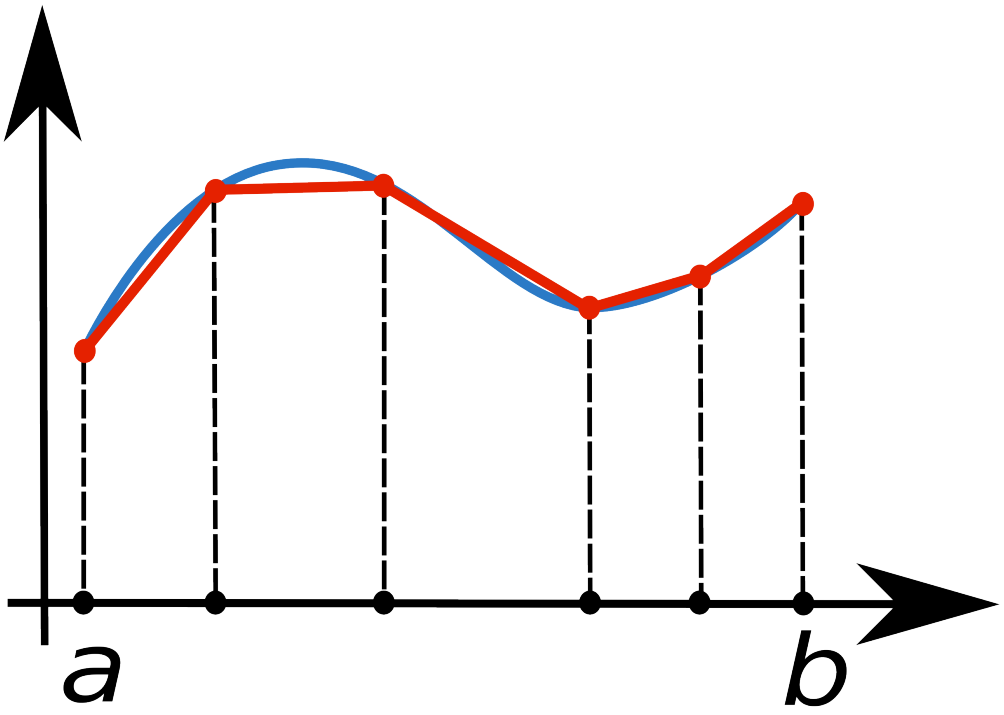
PARALLEL INTEGRAL APPROXIMATION

Using the Trapezoidal Method and Open MPI



Tim Whitaker
CSCI 551 - Spring 2015

PARALLEL INTEGRAL APPROXIMATION

Using the Trapezoidal Method and Open MPI

Tim Whitaker
CSCI 551 - Spring 2015

Introduction

In this project, we will be approximating an integral (that is finding the area under the curve) of a a complex function using the trapezoidal method. The function, the actual value, and the shape of the graph are as follows:

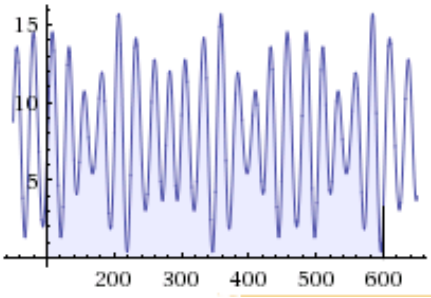
Definite integral: Fewer digits More digits

$$\int_{100}^{600}\left(\cos\left(\frac{x}{3}\right)-2\cos\left(\frac{x}{5}\right)+5\sin\left(\frac{x}{4}\right)+8\right)dx =$$

$$4000+10\sin(20)-3\sin\left(\frac{100}{3}\right)-10\sin(120)+3\sin(200)+20\cos(25)-20\cos(150)\approx$$

4003.720900151326826592913411332738694915902588935612430913165808695414...
995201869556096606327512871082 38

Visual representation of the integral:



As we can see, the graph is quite complicated and the trapezoidal rule will have a lot of opportunity for error. This means we will need to find a value n (for the number of trapezoids) that can accurately approximate this integral. The precision we are shooting for is 14 digits of significant figures. I will write a serial program to find this value of n so we can execute our parallel program using this value.

Finding the Number of Trapezoids

Since the accuracy and error can change so much by even adding 1 more trapezoid, I decided to approach this problem by testing blocks at big increments. So starting at 1 trapezoid, I tested all the areas from 1-100. Then I incremented by a large number (100,000) and tested the next block: 100,001-100,100. I repeated this until my program found an error that was less than 5e-15. I felt that this brute force method of testing ranges at big intervals will give me a better chance of finding a low number that will work, and not take forever to process by testing each and every number starting at 1. I experimented with different ranges and increments and eventually found block ranges of 100 and big increments of 100,000 to be a good combination. All of my files will be attached at the end of this report.

After running for maybe 2-4 minutes, my program ended up with the following values.

```
Number of Trapezoids: 400,090
Found Value: 4003.7209001513379008
Relative True Error: 3.79e-15
```

Using Parallel Programming to Approximate the Integral

In this part of the project, I am using an implementation of the Message Passing Interface called Open MPI. Open MPI can be used with different programming languages, namely C, C++, and Fortran. I decided to program in C++ as that is what I'm most knowledgeable and comfortable with. There are a few different ways to total the sum from many different processes. One way is to use global communications by having processes broadcast their local sum to everyone, and having a root process reduce it all. I chose to do it by having a root process tell the other processes to calculate a part of the total sum. Then each process will send it's local sum while a root process receives it and then totals it all up.While broadcast and reduce may be more elegant, using send and receive calls made more sense to me and seemed easier to debug, so I went with that method.

To run my program in parallel, I am using 10 different computers in our computer lab. These computer assignments are as follows:

```
o251-36 slots=1
o251-01 slots=1
o251-02 slots=1
o251-03 slots=1
o251-04 slots=1
o251-05 slots=1
o251-06 slots=1
o251-07 slots=1
o251-08 slots=1
o251-09 slots=1
```

The slots parameter refers to the number of processes on each computer we are going to use. For the first run we will use just 1 process for each computer. For our next runs, we will use 2, 8, 14, and 20 processes.

In order to time our programs and compare them, I am using a function called MPI_Wtime(). I call this function at the beginning and end of my program (using only the root process to do the timing and excluding all input/output) and take the difference between these times to be the runtime.

Here's a table of my timing runs in seconds.

T1 = 1.64E-01 seconds

#of Slots

1	2	8	14	20
2.03E-02	1.07E-02	9.89E-03	7.44E-03	9.01E-03
2.01E-02	1.08E-02	8.70E-03	1.03E-02	7.96E-03
2.05E-02	1.06E-02	8.80E-03	7.97E-03	1.00E-02
2.02E-02	1.06E-02	8.69E-03	1.17E-02	9.35E-03
2.03E-02	1.07E-02	8.85E-03	8.03E-03	1.08E-02

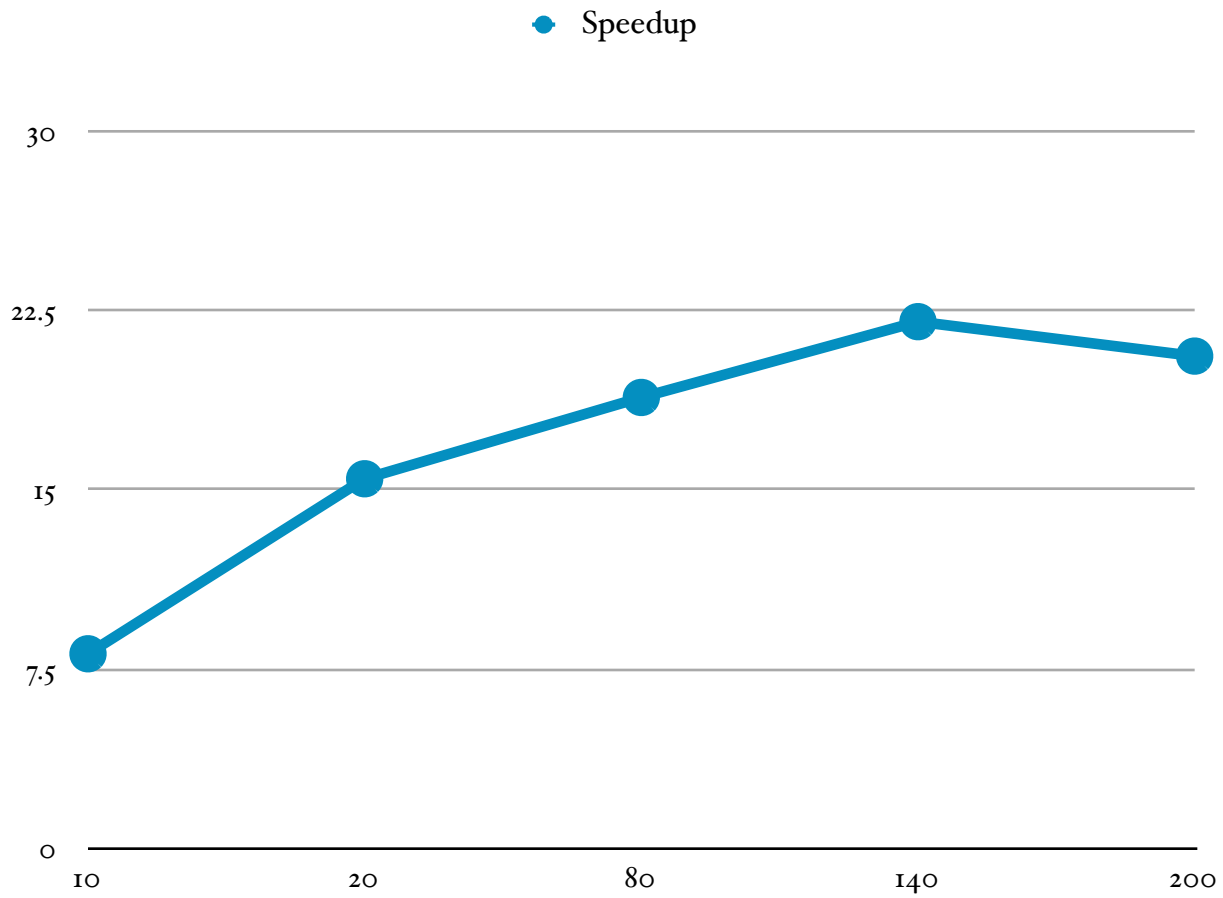
Speedup

Speedup is defined as the ratio between the time it takes to do a task with 1 processor to the time it takes to do the same task with multiple processors. The speedup for this project looks as follows.

Where T1 = 1.64E-01

Speedup = (T1/Tp)

1	2	8	14	20
8.15	15.47	18.87	22.04	20.60

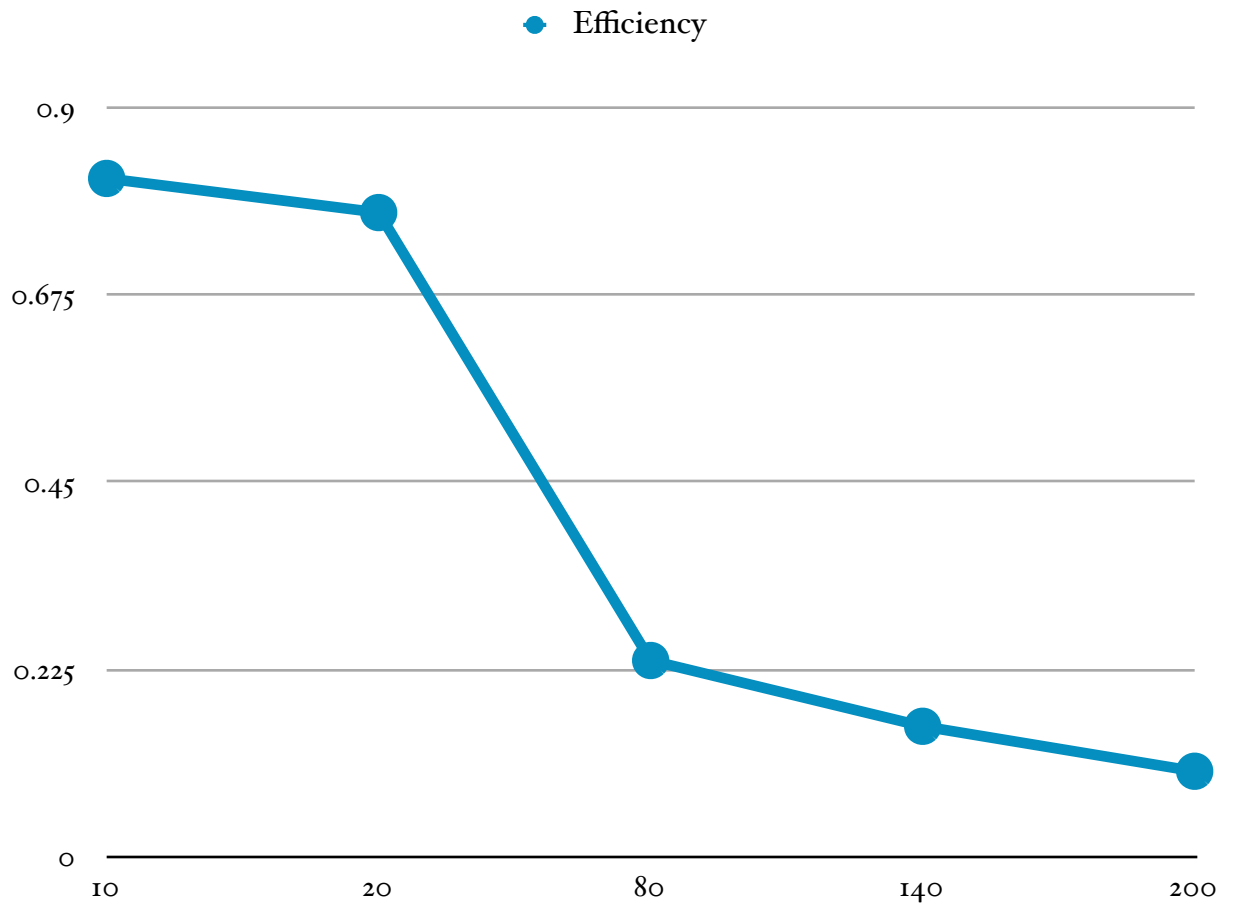


Efficiency

Efficiency is the ratio of the speedup divided by the number of processors. While our slot numbers are equal to 1, 2, 8, 14, and 20, we calculate efficiency by the number of processors. Because we are running these on 10 different computers our process count is 10, 20, 80, 140, and 200. The efficiency for this project is as follows:

Efficiency = (SU/p)

10	20	80	140	200
0.815	0.774	0.236	0.157	0.103



Conclusion

This assignment was a great introduction into the world of Open MPI. Approximating an integral like this gave us a great place to practice Foster's Methodology; Partition, Communication, Aggregation, Mapping. We can see through the timing runs that adding more processors definitely sped up our program. There's a point though, where this speedup drops off. This is because there will always be some bottlenecks that deal with communication and aggregation that can't be improved like calculations and data processing can. This corresponds to the efficiency values we calculated. We can see that as the number of processors rise, the efficiency trends towards 0. I did notice that as the number of processors went higher, our relative true error would increase. I imagine this is because of floating point precision being lost by adding ever smaller numbers with large numbers. The more we chop up the area under the curve, the smaller some of our little values get, which leads to more loss in precision.

Extras

For some extra experimentation, we're going to explore scaling our problem size with our number of processes. For each run, the number of trapezoids we use will be 200,000 times the number of processes we use.

10 processors => 200,000 trapezoids.

20 processors => 400,000 trapezoids.

80 processors => 1,600,000 trapezoids.

140 processors => 2,800,000 trapezoids.

200 processors => 4,000,000 trapezoids.

In order to calculate scaled speedup, we will need to get the time it takes for 1 process to do each set of trapezoids.

T1 = 3.23E-02 seconds

T2 = 6.44E-02 seconds

T3 = 2.56E-01 seconds

T4 =4.50E-01 seconds

T5 = 6.43E-01 seconds

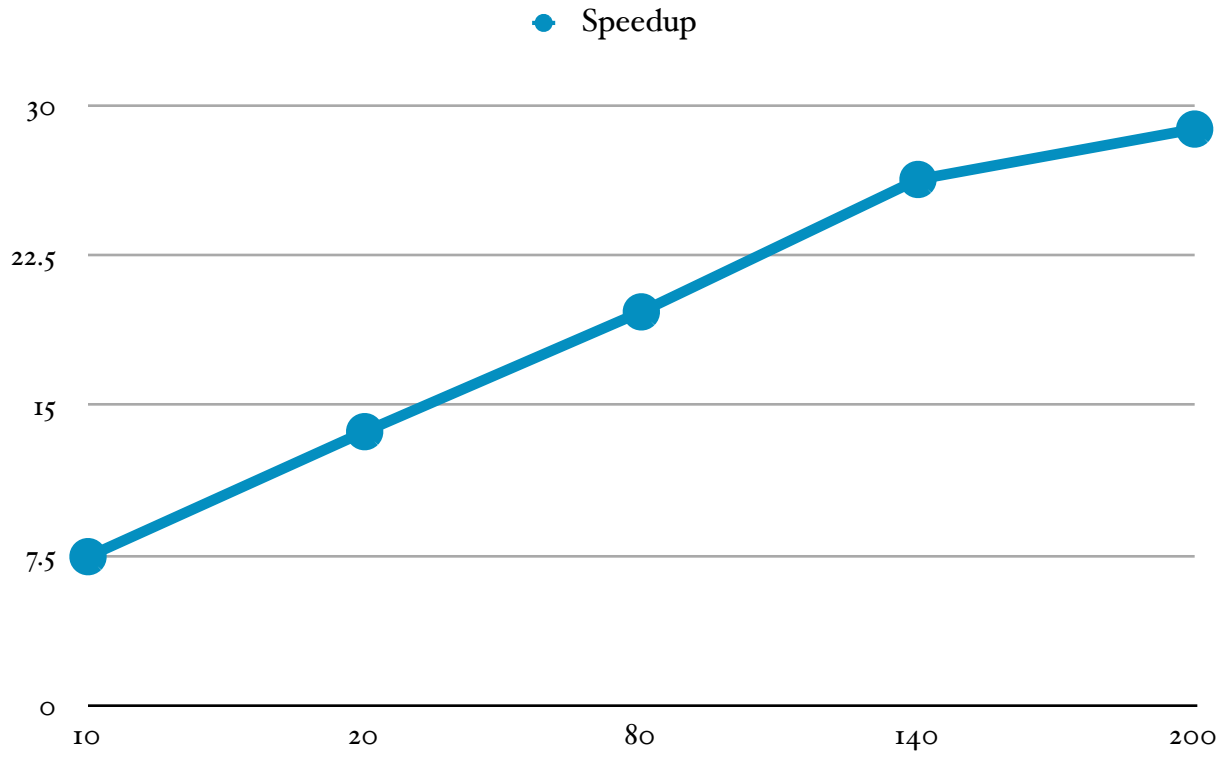
Here's a table of the timing runs from this scaled experiment.

# of Processes

10	20	80	140	200
4.58E-03	4.79E-03	1.30E-02	1.71E-02	2.53E-02
4.44E-03	4.82E-03	1.30E-02	2.29E-02	2.90E-02
4.54E-03	4.88E-03	1.36E-02	2.44E-02	2.74E-02
4.33E-03	4.70E-03	1.31E-02	2.03E-02	2.78E-02
4.46E-03	4.72E-03	1.69E-02	2.31E-02	2.23E-02

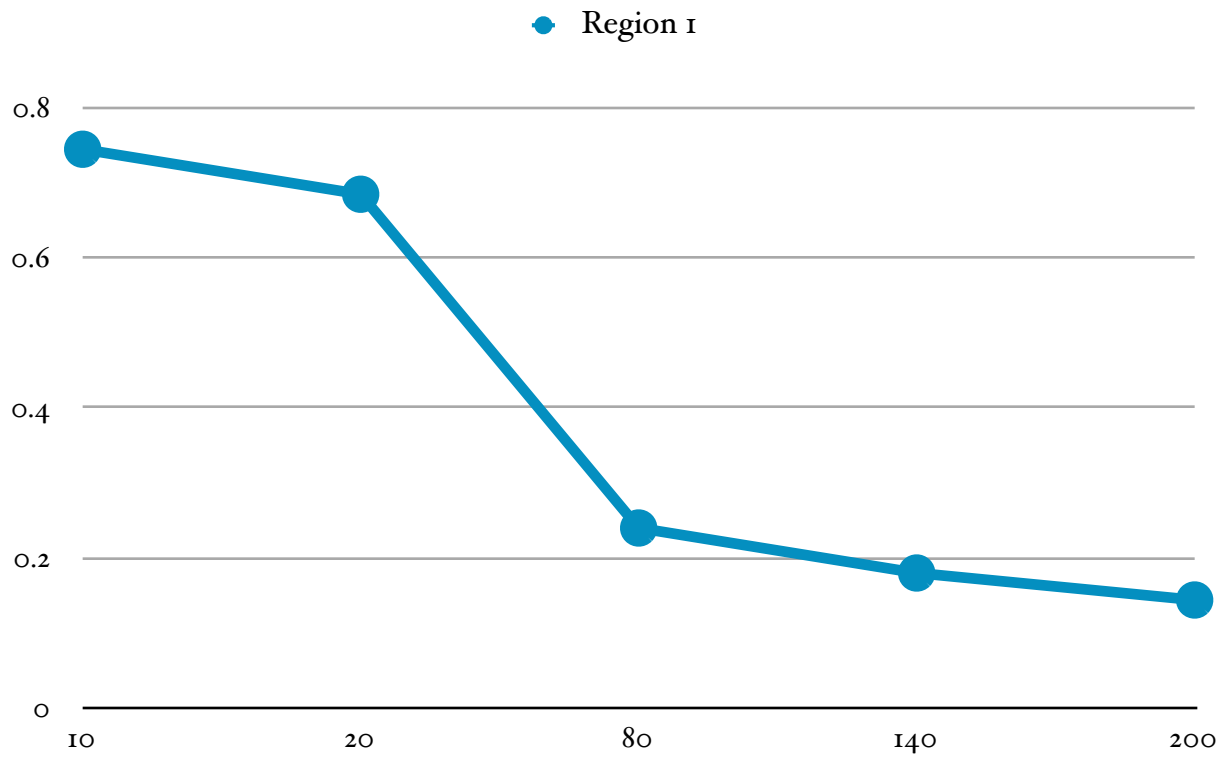
Scaled Speedup

10	20	80	140	200
7.45	13.70	19.69	26.31	28.83



Scaled Efficiency

10	20	80	140	200
0.745	0.685	0.24	0.18	0.144



Scaled Findings

We can see that by scaling our problem size with our number of processors, we achieve a lessened loss in efficiency and a greater gain in speedup. This technique doesn't apply to all problems, but when it can be utilized, it can be quite helpful.