John Olgin
olginj@oregonstate.edu
CS 475 – Spring 2019
Project #2
OpenMP: Numeric Integration

Description

The graphs and raw data in this report are an analysis of computing performance of my program calculating the volume between two Bezier surfaces. My program was developed on macOS Mojave version 10.14 and tested on the flip server provided by Oregon State University. No testing was done on macOS since OpenMP isn't supported.

The program vertically partitions the area between the two surfaces. It then loops through all the nodes summing up the volumes of each partition while taking into account edge and corner nodes. An OpenMP pragma is used on the for loop that utilizes reduction. This allows the program to utilize multiple threads and reduce the input into my **sumVolume** variable. In other words, the reduction command instructs all threads to calculate the volume of each respective node and add their individual values into the final variable **sumVolume.** The **fullTileArea** variable is included in "shared" since it isn't defined within the scope of the pragma for loop. This gives the loop access to the variable.

To track the performance effects of modifying the node and thread variables, I determined the mega heights per second calculated by the program for each combination of thread and node values. For the sake of confidence and consistency, I forced the program to run each combination 100 times and only collected the maximum performance value of all the runs. Mega heights per second were calculated as follows:

$$\textbf{Mega Heights/s} = \textbf{nodes}^2 \textbf{ / (execution time) / 1000000}$$

Volume

My calculated volume had insignificant variation even when altering thread and node counts greatly. It always remained with the range of **28.6875 and 28.7468 or roughly ~29**. Below is the volume calculation for every combination of nodes and threads.

Nodes

Threads	5	10	50	100	250	500	1000	1250
1	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875
2	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875
4	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875
8	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875

12	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875
16	28.7468	28.701	28.688	28.6876	28.6875	28.6875	28.6875	28.6875

You can see the volume seems to decrease very little as the number of nodes increases. However, since the volume seems to be converging on **28.6875** as the node values get higher, I'm inclined to believe it's closest to the actual volume. I assume that significantly increasing the nodes leads to higher precision and a better overall estimate of volume.

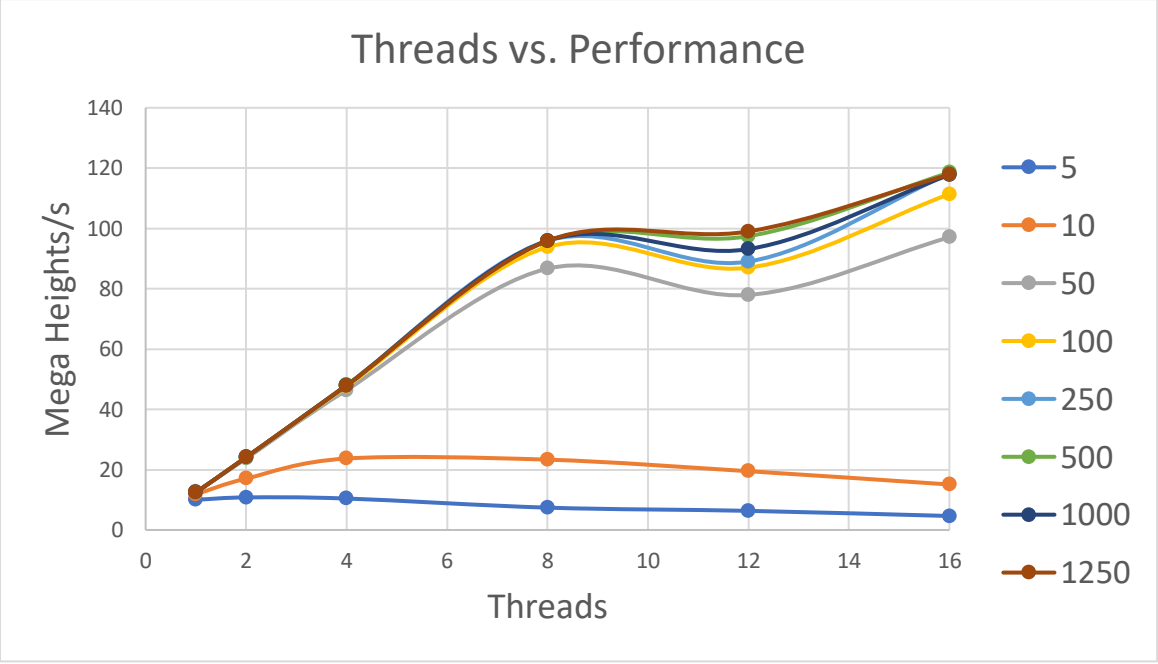
Raw Data

Below is the raw data for the Threads vs. Performance and Nodes vs. Performance graphs below.

Nodes

Threads	5	10	50	100	250	500	1000	1250
1	10.0162	11.8482	12.6621	12.484	12.4846	12.4682	12.4174	12.4049
2	10.7892	17.0233	23.6719	24.0034	24.0475	24.0554	24.127	24.1648
4	10.4328	23.6977	46.3976	47.5582	47.9802	48.0466	47.9376	47.9472
8	7.4092	23.3219	86.6634	93.8315	95.8975	95.8811	95.9302	95.8847
12	6.3467	19.4889	77.952	86.9853	89.0237	97.4282	93.146	99.0014
16	4.633	15.0785	97.064	111.3748	118.2185	118.5002	117.763	117.8687

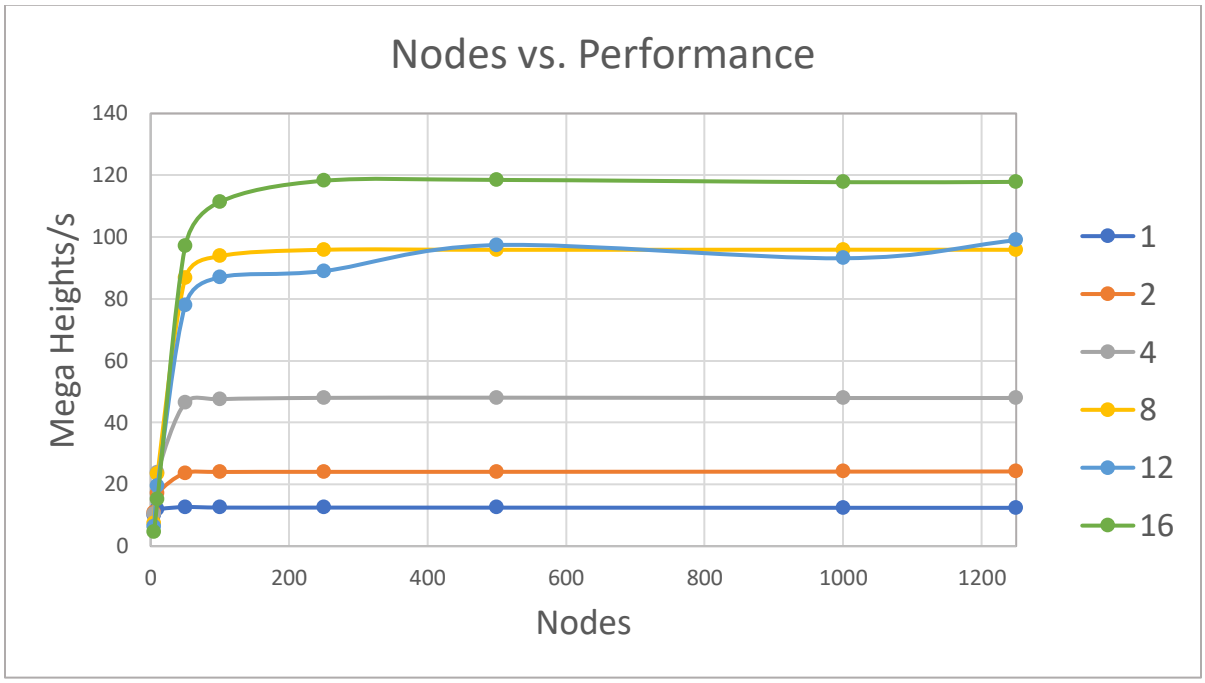
Thread Variation



Generally, the speeds for each subdivision (node count) increased as thread count is increased. However, the 5- and 10-node subdivisions only saw a marginal increase in performance between 1 and 4 threads. Then, a slow, steady decline occurred between 4 and 16 threads. This is most likely due to the problem's smaller size and lack of complexity. While they may benefit slightly in lower thread counts, any additional threads require more setup time. As thread counts climb, this setup requirement will further overshadow any benefit from multithreading, and eventually overtake it. This happens since the problem isn't large enough to benefit from being broken down into multiple threads. Consequently, performance decreased as threads increased.

On the other hand, the remaining subdivisions are large and complex enough to benefit from almost every thread increase (up to 16 threads). However, they all have a small dip in performance at 12 threads. The cause of this could be similar to the issue with the 5- and 10-node subdivisions. Specifically, the required setup time for incrementing threads between 8 and 12 could be completely counteracting the benefits of additional threads within that range.

Node Variation



From the graph we can see that performance remained consistent after 50 nodes for essentially all thread subdivisions. The only significant increase is within the 5- and 50-node range. Again, this is most likely due to problem size and complexity. The 5- and 10-node region doesn't provide enough size and complexity to benefit much from multithreading. As mentioned before, much of the multithreading benefit is counteracted by the setup requirements of additional threads. However, when the node value increased to 50, there was a sharp increase in performance. This was where the benefit from multithreading quickly surpassed the extra work required to setup the additional threads. It can be attributed to the exponential growth in the problem's size as nodes increase. Performance then remains consistent up to 1250 nodes, indicating that node variation had almost no effect on mega heights/s calculated after 50 nodes.

Therefore, I concluded that thread variation has a stronger effect on performance than node variation as the number of nodes increase.

Parallel Fraction and Max Speedup calculation

Calculations based on 1 to 12 threads for 1250 nodes

Speedup

S = (mega heights/s for 12 threads) / (mega heights/s for 1 threads)
S = 99.0014 / 12.4049
S = 7.9808 ~= 7.98

Parallel Fraction

F_p = (NUMT/(NUMT – 1))*(1 – (1/S))
F_p = (12/(12 – 1)) * (1 – (1 / 7.98))
F_p = 1.09 * .8747
F_p = .9534 ~= .95

Max Speedup

S_{max} = 1 / (1 – F_p)
S_{max} = 1 / (1 - .95)
S_{max} = 20