Linge Ge - CS475
Project 1 - Numeric Integration with OpenMP

Overview:

In this project, I wrote a program that sets up two Bezier surfaces. Given a number of subdivisions, the xy-coordinate area of the surfaces is broken up into square tiles based on the subdivision value. Each square tile's volume is calculated as height between the two surfaces multiplied by the tile area. The volume from all tiles are aggregated together, and the final volume between the two surfaces is calculated. This project explores the use of parallel programming to divide the work of calculating all the tiles' heights.

1. The code ran on:
    - Linux os1.engr.oregonstate.edu 3.10.0-693.11.1.el7.x86_64 #1 SMP Mon Dec 4 23:52:40 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
    - 64GiB
    - 32 CPUs @ Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz

2. Based on running the method to calculate the volume, my best guess is that the volume lies around 25.3125 $units^2$. The reasoning behind this is how the volume becomes increasingly precise to pointing at this value as NUMNODES is increased:

NUMNODES	Volume
100	25.313305
200	25.312687
300	25.312595
400	25.312546
500	25.312492
600	25.312515
700	25.312546
800	25.312580

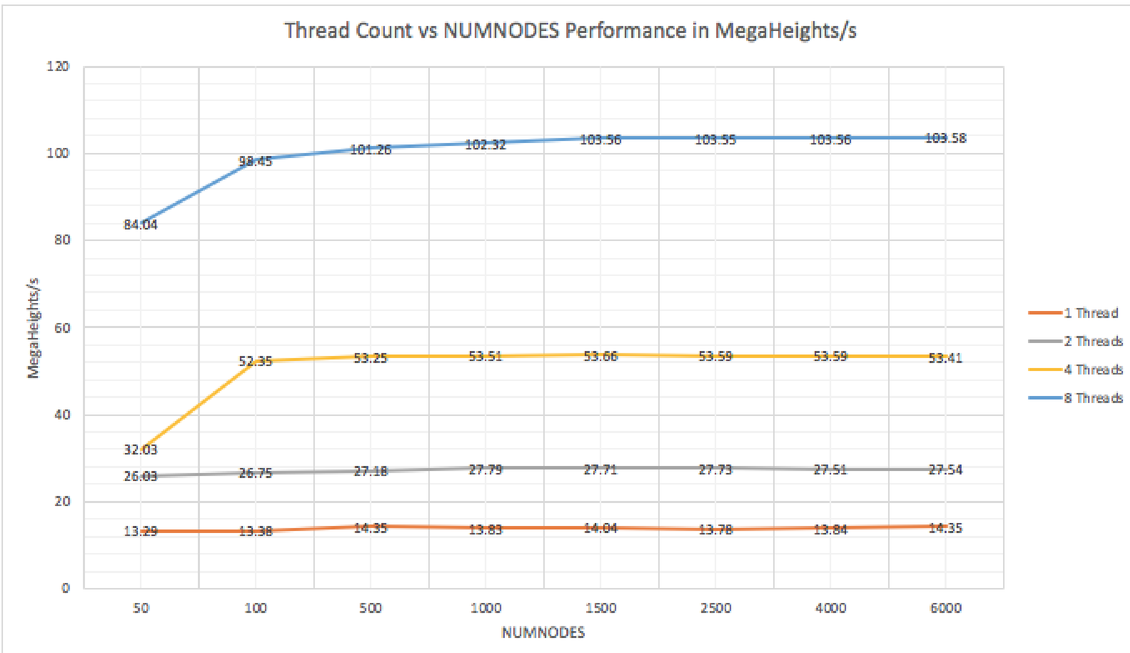
Volumes given by NUMNODES>1000 give less precise values, and this makes sense: given that the subdivisions are precision point float values, we have such a ridiculous small number of subdivision over only a 3x3 grid that at some point such minute precision can't be captured any longer.

3. Below is the master table of all performance metrics collected. The average is taken after running at the same thread count and subdivisions 10 times.

Thread Count (NUMT)	Subdivisions (NUMNODES)	Max MegaHeights/s	Average MegaHeights/s
1	50	13.29	12.85
1	100	13.38	13.21
1	500	14.35	14
1	1000	13.83	13.8
1	1500	14.04	13.78
1	2500	13.78	13.71
1	4000	13.84	13.75
1	6000	14.35	14.12
2	50	26.03	23.91
2	100	26.75	25.82
2	500	27.18	26.81
2	1000	27.79	27.52
2	1500	27.71	27.47
2	2500	27.73	26.9
2	4000	27.51	27.06
2	6000	27.54	27.26
4	50	32.03	26.24
4	100	52.35	49.15
4	500	53.25	51.54
4	1000	53.51	53.29
4	1500	53.66	53.3
4	2500	53.59	53.32
4	4000	53.59	51.23
4	6000	53.41	53.21
8	50	84.04	74.07
8	100	98.45	90.21
8	500	101.26	99.75
8	1000	102.32	100.72
8	1500	103.56	102.82

8	2500	103.55	101.22
8	4000	103.56	98.98
8	6000	103.58	102.85

The performance is plotted in the table below by thread count and subdivisions. Max megaHeights/s is used as the performance unit, as the maximum rather than the average indicates the most accurate value among all the runs.



4. Three major patterns:
   - With more threads, the performance gets better, which makes sense. The work is being actively done concurrently, hence getting finished faster. The performance is also a very precise multiplier: performance is almost 200% better at 2 threads, almost 400% better at 4 threads, and almost 800% better at 8 threads compared to single thread.
   - The number of NUMNODES doesn't affect the rate of performance. This makes sense. Obviously having more NUMNODES means more work, and the program will take longer to finish, but this should not affect performance which is being measured as MegaHeights/s. The fact that differing NUMNODES report roughly the same performance across each thread count is actually a good thing: it shows good precision of the data on repeated runs.
   - For 4 and 8 threads, there seems to be a weak performance dip than expected at lower NUMNODES, which is not seen with 1 or 2 threads. The best hypothesis for this behavior is perhaps that with 4 or 8 threads, there may have been some significant overhead that contributed to overall performance at lower

NUMNODES. At higher NUMNODES, this overhead still exists but is a smaller portion of the overall work that needs to be done.

5. See #4 for behavior explanation.

6. Since the time elapsed was recorded for all runs, F_p can be found by the following equation:

$$\frac{n}{(n-1)}\frac{T_1 - T_n}{T_1}$$

Given how the amount of parallelized worked we actually do varies, the F_p will actually be slightly different depending on the NUMNODES. The following table shows the F_p calculated for each thread at each particular NUMNODE value:

	Baseline		Fp (2 threads)		Fp (4 threads)		Fp (8 threads)
NUMNODES	1 Thread Avg Time (s)	2 Thread Avg Time (s)	2	4 Thread Avg Time (s)	4	8 Thread Avg Time (s)	8
50	0.000195	0.000108	0.8923076923	0.000107	0.6017094017	0.000049	0.8556776557
100	0.000757	0.000389	0.9722589168	0.000208	0.9669749009	0.000122	0.9586714474
500	0.017861	0.009326	0.9557135659	0.004855	0.9709049512	0.002508	0.982379806
1000	0.072486	0.03635	0.9970477058	0.018767	0.9881264428	0.009931	0.9862791239
1500	0.163334	0.081909	0.9970367468	0.042213	0.9887388215	0.021883	0.9897405667
2500	0.455848	0.232359	0.9805417595	0.117211	0.9904968323	0.061827	0.9878505868
4000	1.163508	0.591297	0.9835961592	0.31323	0.9743843618	0.163386	0.9823710464
6000	2.550076	1.320692	0.9641940083	0.676553	0.9795906736	0.350042	0.9859802498

7. The theoretical maximum speedup can be calculated using Fp as 1/(1-Fp). The table below shows the S_{max} given each of the previously calculated F_p calculated in #6. Some patterns:

- Speedup is much lower at lower NUMNODES. This makes sense since the lower the NUMNODES value, the less there are of parallelizable work, and the bigger non-parallelizable work dominates.
- Although the Fp does not seem to vary a lot, a small difference in Fp above 0.97 leads to massively different Max Speedups. This makes sense given the nature of the equation, as the closer Fp gets to closer to 1, Max Speedup jumps quickly.
- The precision among the Max Speedup across each value of NUMNODES for each NUMT seems low, and this is understandable, as the original time data were taken as averages. Perhaps a better precision can be achieved in the future by taking only the time elapsed of the best performing run.

NUMNODES	2-Thread Fp	2-Thread Max Speedup	4-Thread Fp	4-Thread Max Speedup	8-Thread Fp	8-Thread Max Speedup
50	0.8923076923	9.285714286	0.6017094017	2.510729614	0.8556776557	6.92893401
100	0.9722589168	36.04761905	0.9669749009	30.28	0.9586714474	24.19634703
500	0.9557135659	22.58027813	0.9709049512	34.37010904	0.982379806	56.753064
1000	0.9970477058	338.7196262	0.9881264428	84.2207591	0.9862791239	72.88164321
1500	0.9970367468	337.4669421	0.9887388215	88.80065241	0.9897405667	97.47127025
2500	0.9805417595	51.39210823	0.9904968323	105.2280702	0.9878505868	82.30850186
4000	0.9835961592	60.96133291	0.9743843618	39.03865253	0.9823710464	56.72486419
6000	0.9641940083	27.92828668	0.9795906736	48.99720756	0.9859802498	71.32794694