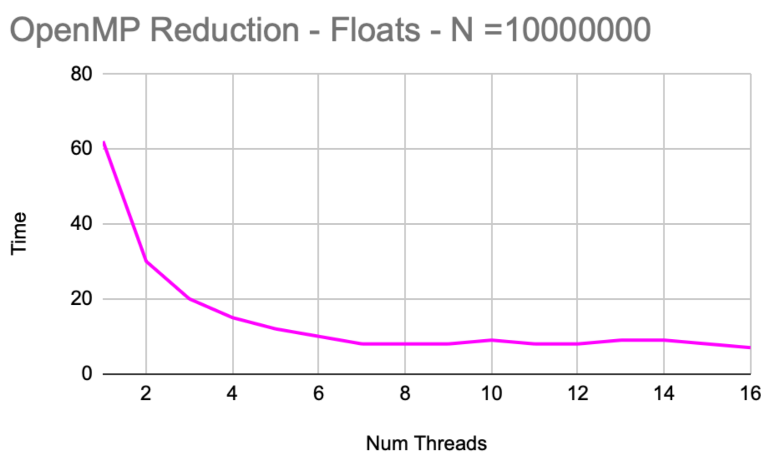
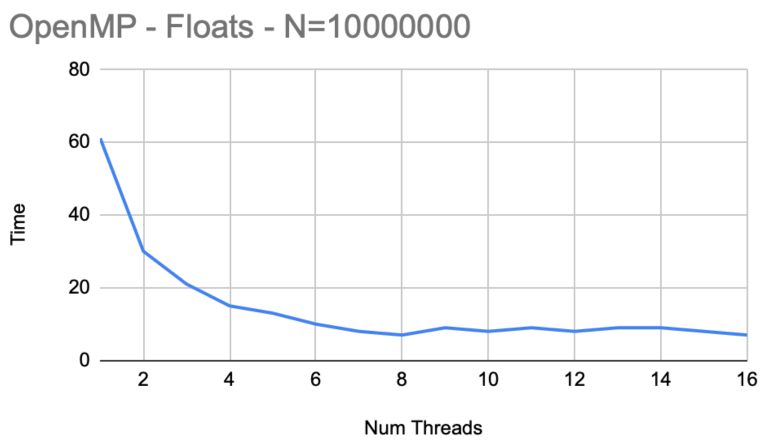
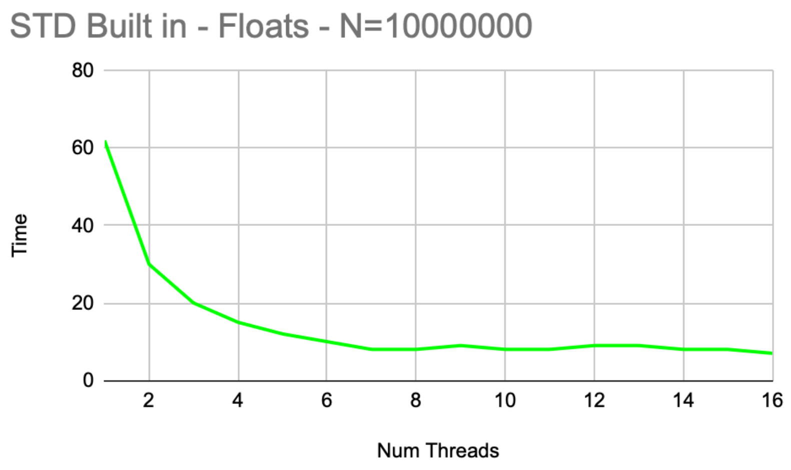
OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024

STRONG SCALING EXPERIMENT 1 RESULTS (floats):

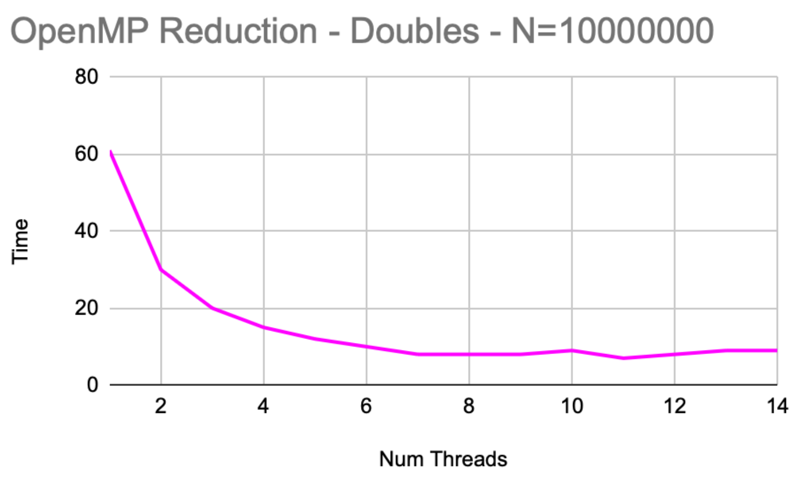


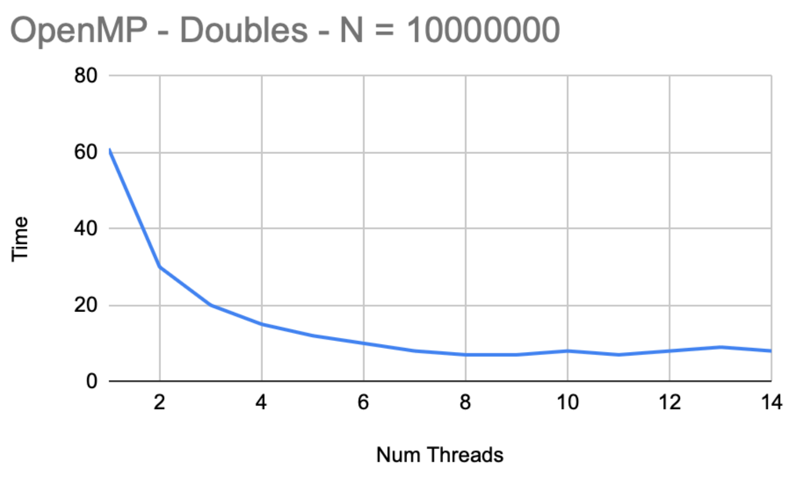


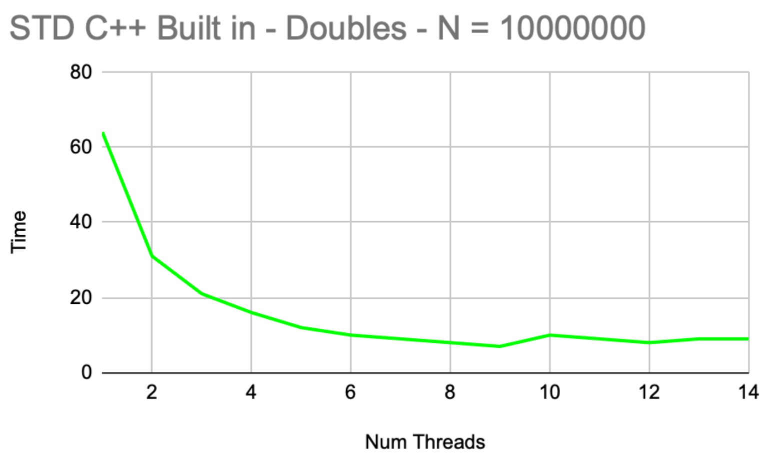


OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024

STRONG SCALING EXPERIMENT 2 RESULTS (doubles):

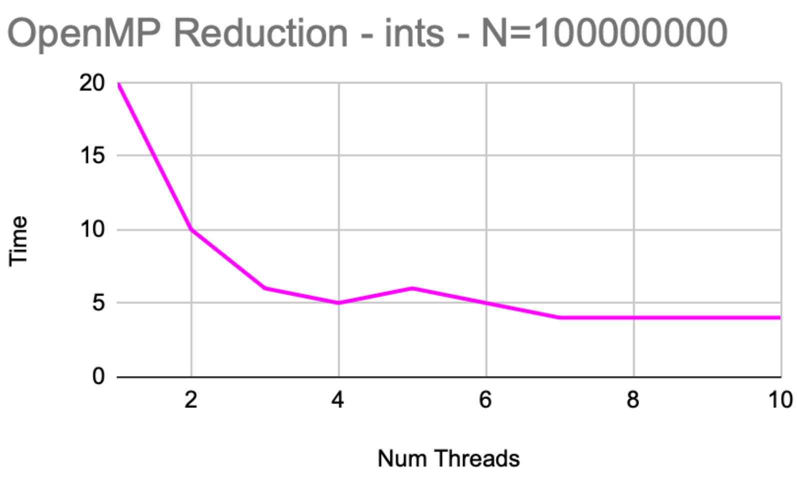


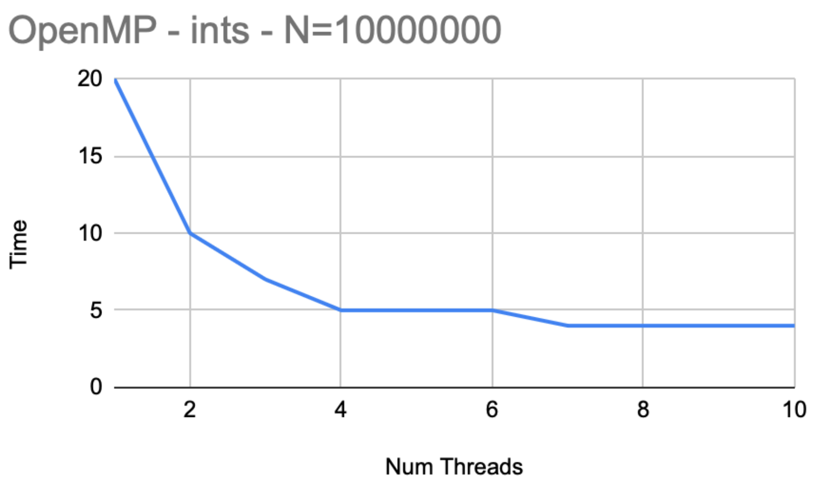


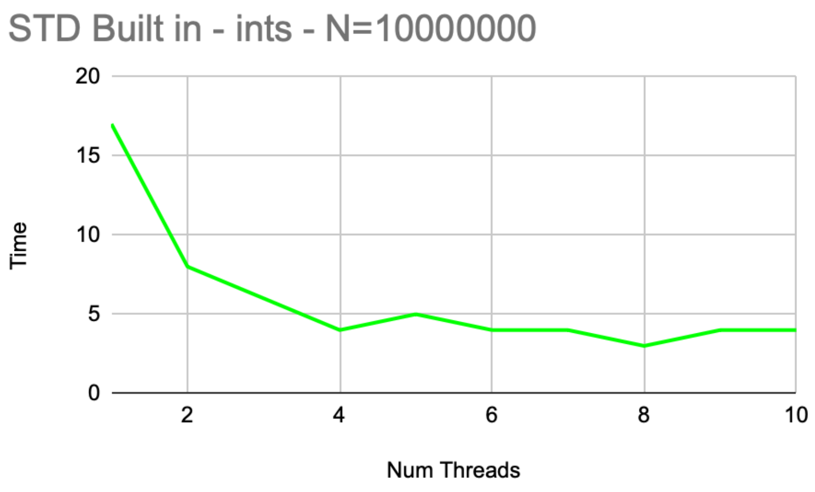


OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024

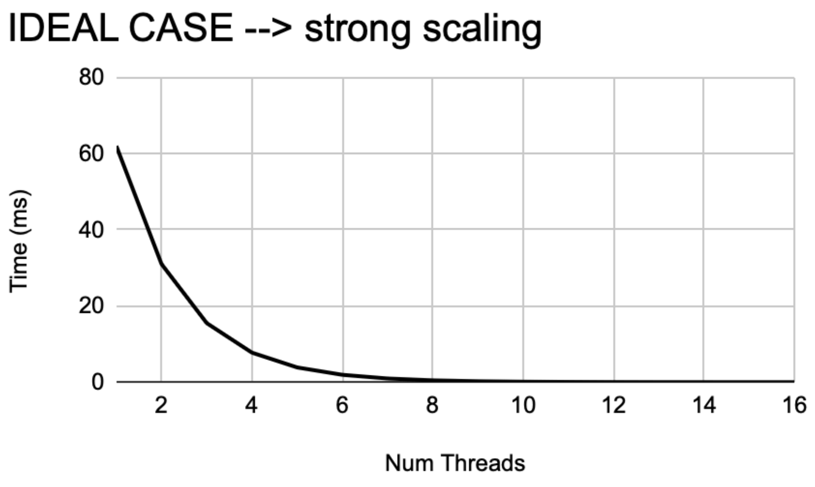
STRONG SCALING EXPERIMENT 3 RESULTS (ints):







OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024



I calculated my strong scaling function for my functions running 10,000,000 elements in an array of floats. I found that when I had one thread for each of the functions, I got either 62 or 61 ms for the time to complete. I then graphed f(x)=62/x (with x being the number of threads) as my ideal case. For doubles, my ideal case was f(x)=61/x. For each of the functions I calculated the Mean Standard Error to see how close these values I observed were to the ideal case (see charts below). For floats, I found that the STD C++ built in was the closest to the ideal case. For doubles, the OpenMP was closest to the ideal case. I am wondering why the OpenMP Reduction was not the closest to the ideal case in either experiment. I think it could be the time it requires my program to go and look for the omp.h file and access its functionality. I had to compile my program using a command on the terminal that told my compiler where to go in my downloads to access the omp.h.

Experiment 1: floats

Mean Standard Error (MSE)	Function
53.18	OPENMP REDUCT
55.11	OPENMP
53.16	STD C++ built in

Experiment two: doubles

Mean Standard Error (MSE)	Function
52.03	OPENMO REDUCT
47.58	OPENMP
58.2	STD C++ built in

I did this MSE calculation for both experiment 1(using floats) and experiment 2(using doubles). Because I got inaccurate outputs for my integer sums, due to integer overflow, I didn't include a statistical analysis of experiment 3 (using ints). I still included the graphs to show that the time did decrease in a similar manner, but it was spitting out nonsense. I could have tried to control for integer out of bounds errors, but I had to create such a large array in order to see any time at all, so I decided to just exclude it from analysis.

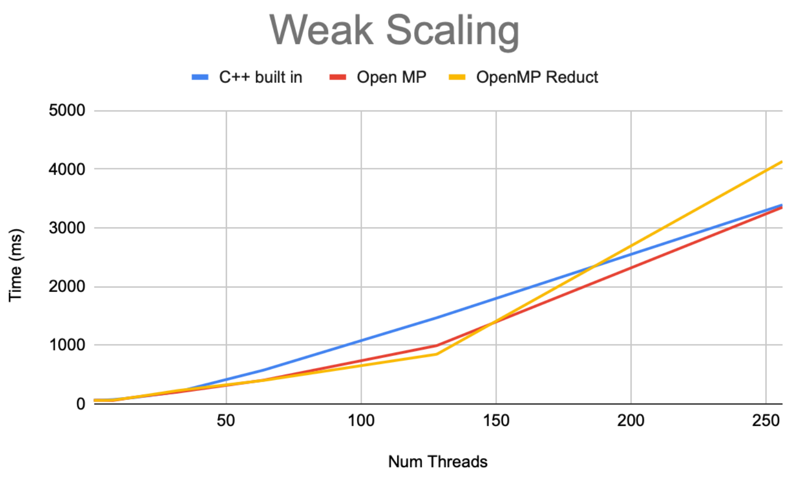
I think all of my mean standard errors are so high because my ideal case goes to zero. While this does follow the theology that doubling threads should half the time it takes to complete work, it isn’t the reality. Ultimately the overhead of creating threads, assigning them work and doing the work will take some time to complete. I don’t think we could get this program to run at that fast of a speed on my mac. We also must remember that my computer has a limited amount of cores, so while I can create 16 threads, they won’t be able to truly run in parallel due to the limits of my computer. If we had a super computer, then maybe we could get these outputs closer to the ideal case.

MSE is also not the most ideal model to look at when comparing my functions. I could have used Amdahl’s law which would control for which parts of the program are parallelizable and which are not.

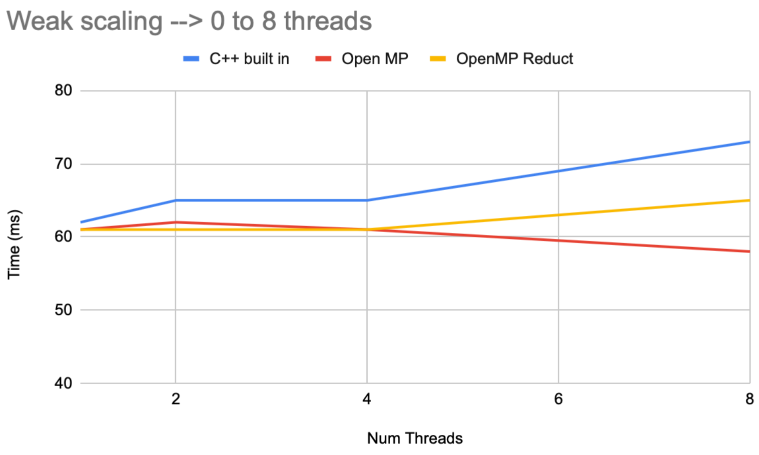
OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024

WEAK SCALING:

To test weak scaling, I doubled the array size (amount of work) and the number of threads with each experiment. I started with 10,000,000 for my array size and then doubled the array size with each run of the experiment. I also doubled the amount of threads for each run. Below are my results. The ideal case would be to keep the time that it takes exactly the same. This would be shown in a graph as a straight horizontal line. I found that all of them did not perform like this. In my experiments, I did go up to 256 total threads.



My computer only has 8 cores. I discovered this by running this command in my terminal: "sysctl -n hw.physicalcpu". I then graphed my experiments as it went from 1-8 threads to see if it followed more of a flat trendline - which would be closer to the ideal case. I found that this was the case and this graph was significantly more flat than the one that went up to 256 threads. From the graph below, it looks like the OpenMP reduct is the best graph in terms of staying flat from 0 to 8 cores. I do think that the OpenMP is also probably good at weak scaling, since it goes down. I believe this could be just due to some extra noise in my program timing.



OpenMP Parallel Reductions Assignment
Samantha Pope
CS6013
04.16.2024

Array Size (N) of floats	Num Threads	C++ built in	Open MP	OpenMP Reduct
10000000	1	62	61	61
20000000	2	65	62	61
40000000	4	65	61	61
80000000	8	73	58	65
160000000	16	109	115	114
320000000	32	206	201	227
640000000	64	578	406	402
1280000000	128	1469	994	847
2560000000	256	3392	3352	4134