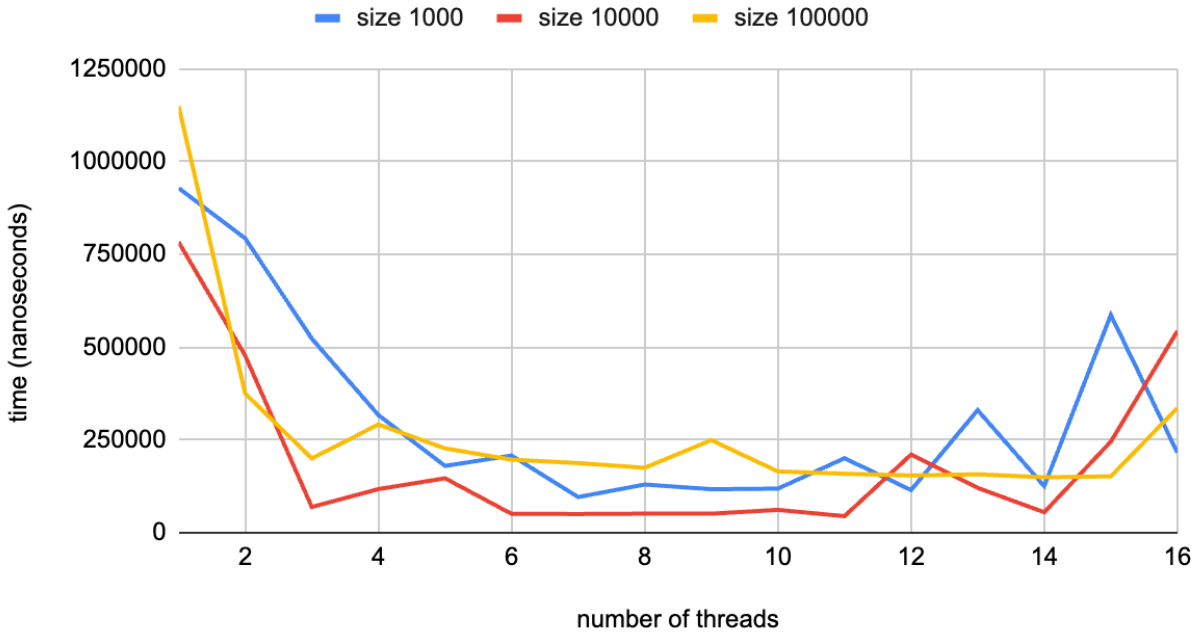
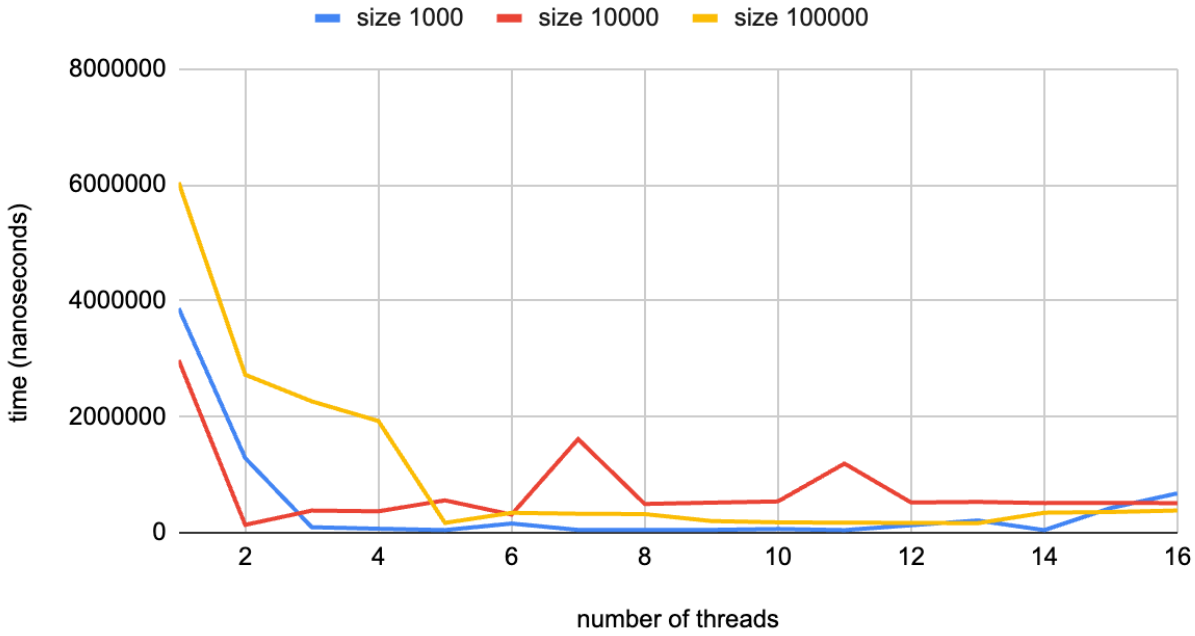
1. **Strong Scaling**: For this test, you will fix the input array size **N** and increase the number of threads **num_threads**. For instance, you set N=1000, then increase the number of threads from 1-16, and plot the resulting time values. This produces one curve. Then, set N=2000, repeat the experiment, get a second curve. Produce a few curves of this sort for each of the three functions. There should be 3 graphs in total, one for each function. Do the curves match the ideal case? If not, why?

Both of the open MP graphs are more in line with the ideal case, because as the number of threads increases, the work is divided among those threads which shortens the execution time. My c++ custom reduction looks wacky though and does not match the ideal case for strong scaling

strong scaling openMP custom reduction

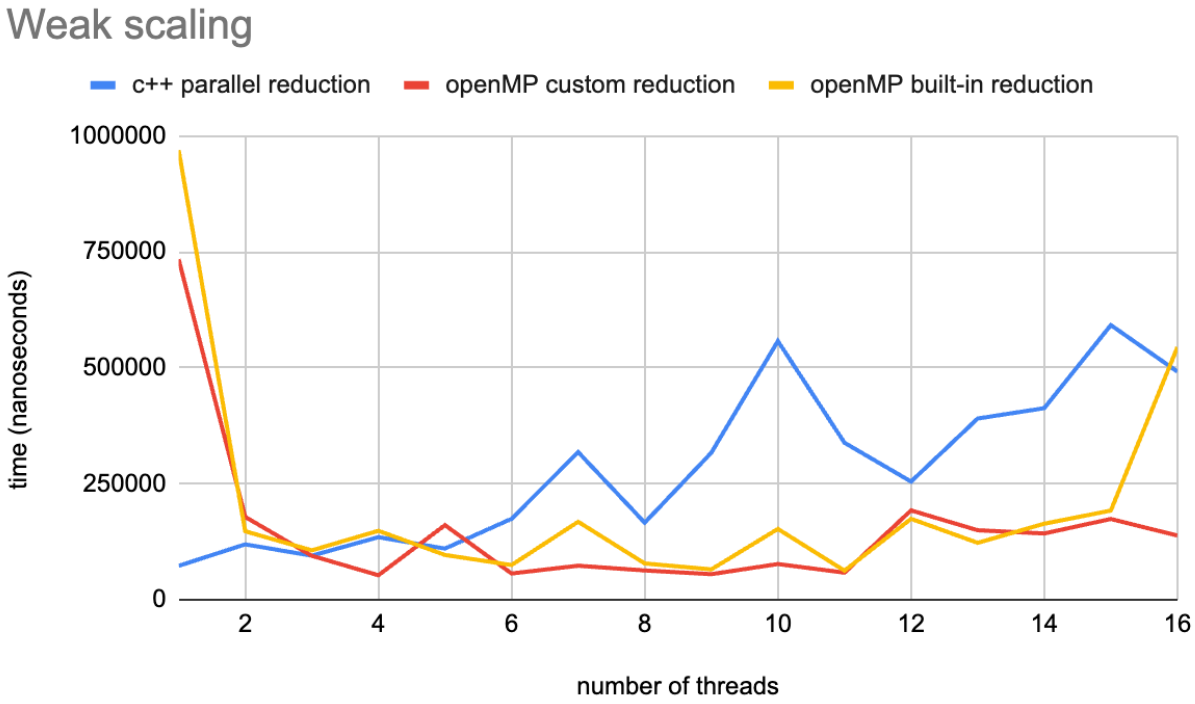


strong scaling openMP built-in reduction



2. **Weak Scaling**: For this test, you will increase **N** and **num_threads** together. Assume the work is divided evenly between threads. Then, the time taken should roughly stay constant as **N** and **num_threads** are simultaneously increased. For each function,.plot a single curve as N and num_threads are increased, with time on the y-axis as above, and either N or num_threads on the x-axis. Produce a single graph containing results from all 3 functions. Does the curve match the ideal case? If not why?

I think it does match to the ideal case because threads are being added as the size is being increased which means the work is being divided but that isn't reducing the amount of work per thread. The custom c++ one still looks a little wacky



3. **Comparison**: Which of the three methods above performs best on each test? Can you explain why?

I think in theory the c++ custom reduction should be the fastest as it would be the most specialized to the task at hand because it can be explicitly written for exactly what it needs to do. The built in should perform the worst because it is the least specialized and is supposed to be generic and easy for people who may not understand multithreading. The custom openMP should be somewhere in between as it is still customized but built on the more generic base of omp that will require a bit more processing time