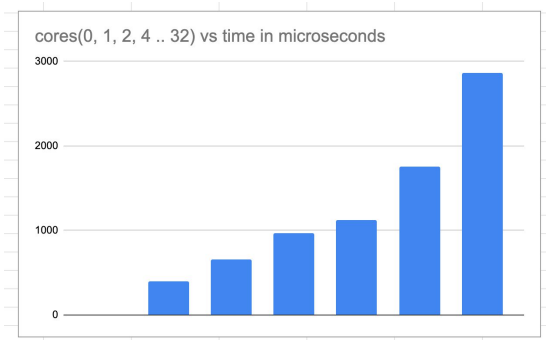
Kous Nyshadham (kn8794)

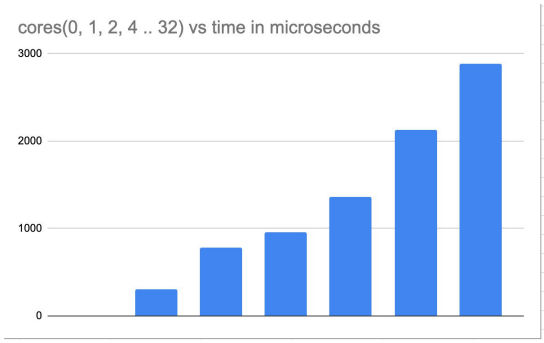
Approach:

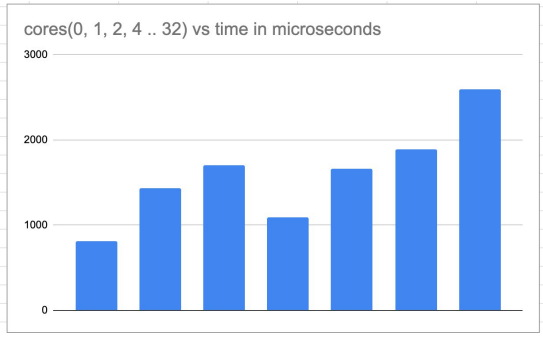
Considering I started about a little less than a week before the due date, I was able to focus on step 1, 2, and 3 independently over the course of many days. Therefore, the first day I focused on just setting up everything (reading input, creating scan-op, trying to use template specialization) and getting the sequential implementation to run. The next day I focused on researching the parallel scan algorithm. I used chapter 1 in the Parallel Programming book by Calvin Lin to get a general high-level understanding of the algorithm I would need to implement and then after reading through many links and tree/array diagrams came upon the Nvidia GPU article which provided nice pseudocode with good explanation. Although I did not understand this at first, I eventually did and wrote a program that pthread created and joined every time for the parallel operations. I eventually improved this by instead of creating and joining pthreads inside the level traversing for loop, created a number of threads equal to the number of threads I decided would ever be used (arr.size()/2 and then making that the nearest power of 2) and then passed into the thread create a function (upsweep and then downsweep after thread creating and joining since two functions and couldn't combine into one) which did the algorithm on its part of the array (left bound = i* (n/createThreads) right bound = i*(n/createThreads)+(n/createThreads)). After reading piazza and realizing that I must create the number of threads given and not decrease, I decided to just set some of the threads to bs left and right values so that when the inner for loop check of the algorithm runs, it won't go through and it will just wait at the barrier outside the inner for loop right before the end of the outer for loop (level traversing for loop). Throughout this whole process, I slowly removed template specialization and just copy pasted two functions for each type of input. After this, I read chapter 6 in the textbook about posix threads, specifically about thread_cond_variables and the barrier implementation and afterwards wrote my implementation which was a combination of the arrive() and wait methods in the book. The barrier part took me a few hours whereas writing the parallel algorithm took me a couple days (many bugs such as me thinking 100000_5000_floats.txt not working for me but only because when I copied the file over only about 2500 multidim vectors being in the file).x

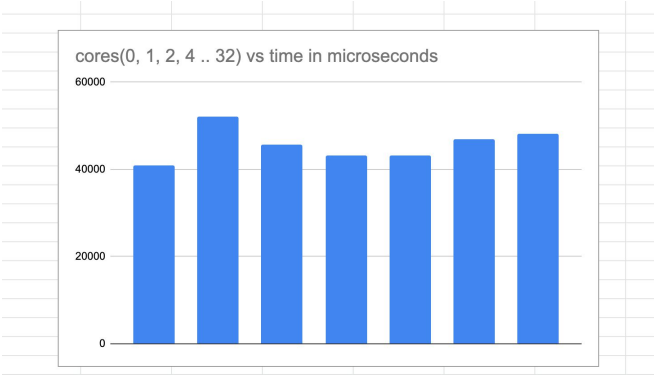
Performance Results:

Before exploring trends in the graph, something to note is that summing integers in parallel has less of a speedup than summing multidimensional floating point vectors in parallel. This is because of how insignificant the addiction operation of integers take and considering that we don't do and operations at once but rather amortize it over the tree reducing the number of operations occurring at once as a consequence of only doing operations at a particular level. Graphs will be in order of left-right top -bottom 10_ints, 5_2 floats, 100000_ints, and 10000_500_floats. If I had allotted more time for generating a script that would generate more inputs and outputs and graphs using jupyter notebook. The x-axis is the number of threads going from 0, 1, 2 … to 32 in multiples of two and the y-axis is time taken in microseconds.









The top two graphs increased with number of threads fairly linearly because of the fact that such a small input size leads to the addition of parallelism adding no extra benefit and only overhead - pthreads. For 100000 integers, the time it takes stays the same initially because while it parallelizes the task, the integer addition is so trivial it makes little to no difference, however once the threads > cores, overhead increases the time. For ten thousand floats with 500 dimensional vectors, the time initially increased because of overhead of creating one pthread vs 0 and then went down by about 10% and leveled off until 4/8. This is because the overhead of creating threads begins to factor in again when the computer does not have as many cores to run the threads in parallel.I believe my implementation of barrier is the same as the implementation of pthread_barrier based on what was covered in chapter 6 and therefore, the performance was exactly the same as expected.

Time Taken:

Sequential took about 3 hours to implement with the hardest part being finding the concept of template specialization (idk c++) and having to restart the moment I finished because in my makefile I had clean: rm main.cpp instead of rm pfxsum. Parallel took about 10 hours (maybe more if you include the researching/learning concept time) because of all the bugs, and barrier took about 2/3 hours (reading ch 6 & online, learning about pthread_cond variables and coding).

Reflection:

Allocate time for data collection and report RIP.