To parallelize the algorithm, I largely followed the instructions in chapter 9 and George's comments from class. The steps were as follows:

- Each process generates N / P elements
- Each process calls the parallel quicksort function
- In parallel quick sort:
    - A pivot isselected by getting the ***mean*** (I discussed with Ansi in office hours) of each process's elements, and then selecting the median of all processes' means. This pivot is then broadcast to all processes
    - Each process makes local less-than-or-equal and greater-than arrays based on the pivot in parallel
    - MPI is used to create a prefix-sum array for each process's local less-than-or-equal and greater-than arrays' lengths.
    - The processes are split based on the proportion of total less-than-or-equal and greater-than elements across all processes.
    - MPI leverages the prefix-sum arrays and number of processes for less-than-or-equal and greater-than to send all less-than-or-equal values to the lower rank processes set aside for less-than-or-equal values, and all greater-than values to the remaining higher rank processes.
    - Once all of the values are in the appropriate processes, the processes are split into new communicators based on whether they are less-than-or-equal or greater-than process, and each communicator recursively calls parallel quick sort.
    - Parallel quicksort is called recursively in this way until the communicator that is passed in only has one rank, at which point the passed array has the C++ standard qsort function run on it, and the sorted array is returned up the recursion tree.
    - To ensure that each process ends with N / P elements, at the top of the recursion tree, a call is made to balance values across processes before returning. This sends values up or down ranks while maintaining sorted order (values passed down are from the front of an array and end up at the end, values passed up are from the end and end up at the front)
    - The now balanced processes are returned.

Timing

The following timing results are the averages of 3 runs each

1M elements

P = 1 0.7871s
P = 2 0.4593s
P = 4 0.318s
P = 8 0.2460s
P = 16 0.1662s

10M elements
P = 1 9.2067s
P = 2 5.2202s
P = 4 2.9516s
P = 8 1.7476s
P = 16 0.9958s

100M elements
P = 1 103.8148s
P = 2 59.2467s
P = 4 32.0970s
P = 8 17.8957s
P = 16 9.9806s