Isa Maguire
Parallel and Distributed Final Report (Sorting)
5/9/23

To improve the speed of sorting, I first created a fork-join pool for multithreaded recursion. I then implemented a recursiveAction to do quicksort on a range of values. My first attempt was just a basic quicksort using Hoare's scheme, which takes pointers at the beginning and end of the range and swaps the values when the one on the right is < pivot and the one on the left is > pivot. I originally chose this method because it has much less swaps than other methods, so I assumed it would be faster. I just chose the pivot as the last value in the array. I also made sure to store the value of the pivot beforehand so that I wasn't accessing the array extra times each iteration.

I also added the base case of an array less than size 1000 being sorted using Arrays.sort(). I played with this value a bit but the program speed decreased when I used 100 and 10000, so I just stuck with 1000.

My runtime for this method on my computer got 2347ms, but on the cluster it was much slower at 3500 ms. Apparently this had to do with how the cluster combined solutions, and using too many threads would actually make the process take longer. To fix this I set the number of threads to be fixed to a smaller number when running on the cluster. I experimented with a few different values but the sweet point was at 16. Going under that put my runtimes too close to the time with 8 processors (my computer) and going too far over that also slowed me down. With 16 processors I got a runtime of ~1650ms.

The next step was choosing a better pivot. A bad pivot causes recursion to go deeper, and I wanted to see if I could find a way to take care of that. I chose to use "middle of 3" because using a random pivot still gives a higher chance of failure. With this my runtime got to around 1550ms which was a bit faster than before.

One thing about Hoare's scheme is that each iteration accesses values at each end of the array. If we have a considerably large array, then our cache will have to keep refreshing different ends of the array which takes extra time. If we had a method that only moved over the array linearly, then our cpu could access the next value without having to update the cache. Lomoto's scheme for partitioning does this, and even though there are more swaps on average here I might be able to see a speedup. I tried Lomoto's scheme and my runtime on the cluster was ~1700, so a bit slower. Next I tried setting it up so that larger arrays(50000>) used Lomoto's and smaller ones used Hoare's. My runtime went to ~1575 but it was more consistent than the previous setup so I used this one instead. I also tried fork/fork vs fork/compute but my fork/compute was faster, so I stuck with that also.

My last improvement was trying to use tail recursion, where I sent the smaller task to fork and the larger one to compute. With this my runtime went under 1500ms. I wanted to further improve this with vectors but unfortunately I ran out of time.