Brian Altamirano
CPSC 479

Github link:
https://github.com/AltBrian379/CPSC-479-Project2

Files Included:
main.cpp, parallelsort.h, parallelsort.cpp, parallel.exe
Compiled using (Windows):
```
g++ -o parallel main.cpp parallelsort.cpp -fopenmp
```

Run program in CMD (Windows)
```
C:\parallel.exe <number of elements>
// OPTIONAL COMMANDs:
// '- p'  prints out the list of elements for both.
```

Run program in Powershell/Visual Studio Terminal
```
./parallel <number of elements>
// OPTIONAL COMMANDs:
// '- p'  prints out the list of elements.
```

Notes:
Threads set in program within "#pragma omp parallel" directive. 16 threads are run in parallel.

The numbers generated are from 0-99. This can be set in the createRandomNumbers function in main.cpp

Somewhere between 10000000 and 2000000 elements causes the program to throw a bad memory allocation.

'-p' takes forever to finish as std::cout is super slow.

Description:
The program consists of 2 cpp files, main.cpp which runs the program and does the sorting and timing of std:: sort and the timing of the parallel sort. Parallelsort.cpp holds the code for the implementation of Parallel Quicksort.

The program accepts a flag to allow for printing the elements and without. Without is favorable as std::cout is super slow on the larger element amount.

We first create the random list of numbers and initialize it to a vector of ints dynamically allocated to the heap. The heap is preferable as the stack has limits on memory and is more restrictive as we will be recursively implementing the parallel quicksort. We then save a copy for use with the std::sort and Quicksort.

We time both sorts and output the result to screen.

In the parallel sort file we have 3 functions. Parallel sort is what the main function calls. recursiveParallelSort is where the meat of Quicksort is. CheckifAllSame is a check to make sure the program doesn't get into a recursive loop.

recursive ParallelSort implements quicksort. The base condition is that if the size of the list given is 1. This is important as this means that we are done breaking down the list as is done in quicksort.
There is a check to make sure that the elements in the list are not all the same. If allowed, then the program will be stuck in a loop where it will continue putting all lesser numbers and itself on the same list of the same size over and over again.

We then initialize the random number generator that will determine our pivot.

We then create 2 lists, 1 left list and 1 right list in the heap.

Then using OpenMP we parallelize the for loop that would populate these two lists using 16 threads. We first create 2 private lists corresponding to left and right. This is to make sure there is no race condition and to benefit with speedup.

Then we go through a for loop with no wait, populating the list with the element that corresponds with the algorithm of quicksort: if the element is less than or equal to the pivot, we put it on the left list, if it's greater than, then it goes to the right list. We then append the private left list (if populated) to the public left list, and the private right to the public right list using "omp critical" to prevent any race condition.

We make sure to prevent any wasted time by always checking if the other list is empty, as if it is, we just return the list. Otherwise we continue with the recursive steps.

Once it returns from recursion, we create a new vector list and put the left list first, then the right list first to combine them, and then return that newly created list that is sorted.

Sample Ouput (./parallel 10 -p)

```
PS C:\Users\Brian\Programs\CPSC-479-Project2> ./parallel 10 -p

Printing is enabled...
Program Starting... input = 10

We are creating 10 random numbers for sorting!
Initial list: 47 17 41 32 36 1 64 87 21 40
```

```
Sorting using std::sort()...
std::sort total time: 0


Sorted list of size 10 (std::sort()): 1 17 21 32 36 40 41 47 64 87
We are starting my Quicksort...
We are ending my Quicksort...
Sorting using Parallel Quicksort
Parallel Quicksort total time: 0.01


Sorted list of size 10 (Parallel Quicksort): 1 17 21 32 36 40 41 47 64 87
```

Sample Ouput (./parallel 100000)

```
PS C:\Users\Brian\Programs\CPSC-479-Project2> ./parallel 10000000
Program Starting... input = 10000000

We are creating 10000000 random numbers for sorting!


Sorting using std::sort()...
std::sort total time: 3.148


Sorting using Parallel Quicksort
Parallel Quicksort total time: 2.337

PS C:\Users\Brian\Programs\CPSC-479-Project2>
```

Psuedocode for parellelsort.cpp

```
parallelSort(list)
{
    If list is size of 1, return the list
    If list is all the same elements, return the list

    Randomly select pivot

    Parallel for through all elements
        If  element less than or equal to pivot
            Add to list left
        If  element greater than pivot
            Add to list right
    parallelSort(left list)
    parallelSort(right list)

    Combine list left and list right

    Return combined list



}
```