Task 1

a) In task a I just added internal counters for each worker, and made them interact with their own cells in a shared array, thus producing mins, maxs and sums for each individual row. the last thread consolidates all of this data, there is no need for locks as there is no truly shared resources.

b) In task b the main change is that I used a global variable to store all the data. That makes locking it necessary. At first I updated the globals with each and every cell to update it. As it turns out it was harming performance a lot, so now the program uses local variables and locks all three at once. It took like 10s beforehand

c) This time a bag of tasks approach is to be used. The tasks are rows, and for that there is only need for one counter that signifies which row is to be processed. Each thread gets the number and then checks if it is still in bound of array, if not it breaks the while, and returns from the thread.

```
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./1a
The total is 604602040
The max is 98
The min is 0
The execution time is 0.012555 sec
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./1b
The total is 604602040
The max is 98
The min is 0
The execution time is 0.013252 sec
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./1c
The total is 604602040
The max is 98
The min is 0
The execution time is 0.011815 sec
```

And this is the execution for the bad logic:

```
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./1_bad
The total is 604602040
The max is 98
The min is 0
The execution time is 3.48649 sec
```

Task 2

In task two I implemented a quick sort algorithm with parallel threads. I opted for a recursive approach in which some of the recursion can be called with a thread instead of executing it on itself.

My algorithm relies on passing a struct which consists of a pointer to an array to be sorted, and the size. This avoids any copying of data between threads.

Each thread first sorts it's array into the bigger and smaller than pivot and then, if there is a worker slot available passes the bigger array of to a worker, to make the thread creation worth it.

After writing this and analysis I have come to the realization that a pool of workers would be more efficient. To offset that I added a condition to not multi thread the smaller arrays if the array being processed is too small (20x smaller than the original array seems to be the sweet spot). This gave me a considerable performance boost, going from 1.2sec to 0.6 sec. Which is 4x faster than a single threaded program at 2.5sec (all the times are at max array size of 1 million [above that my recursive implementation segfaults]).

```
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./2 1000000 1

The execution time is 2.50542 sec
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM1$ ./2 1000000 10

The execution time is 0.669254 sec
```