Compile:
For bubblesort:
make bubblesort
./bubblesort [size of array] [number of threads] [optional -s flag to also run sequential]

Example;
make bubblesort
./bubble 100000 4 -s

For mergesort:
make mergesort
./mergesort [size of array] [number of threads] [optional -s flag to also run sequential]

By default the programs will always run and output time taken to do parallel version. Adding the -s flag with make it also run the sequential version.

I ran and tested code on the ilabs.
The ilabs can be laggy sometimes. I can run a piece of code and continuously get 40s but if i run later in the day i get 10-15 secs.

There is variance in the data because some arrays are better optimized than others.for different sorting algorithms. I ran multiples tests, dropped extreme outliers, and these were the results.

Mergesort-

units	seq	par	4 threads	8 threads
10000	1-2ms	1-3 ms	0-2 ms	0-2 ms
25000	3-5 ms	3-8 ms	1-3 ms	1-3 ms
100000	7 ms	7-8 ms	4-9 ms	6-9 ms
250000	36 ms	19-23 ms	17 - 19 ms	13-19 ms
500000	77 ms	40-44 ms	34 ms	28-35 ms
1000000	150-170 ms	82-89 ms	70 ms	62 ms
30000000	7-10 sec	2 s 850 ms	2 s 189 ms	1 s 877 ms

Bubblesort--

Some values can be much larger than listed, but on average of 10 runs this was the range excluded worse case outliers.

units	seq	2 threads	4 threads	8 threads
1000	3 ms	4-7 ms	3-5 ms	3-6 ms
25k	1 s 498 ms	1 s 224 ms	1s 118ms	1s 114ms
50k	6 ss 199 ms	4-6 s 761 ms	5s 137 ms	3 s 417 ms
100k	26s 184 ms	23 s 341 ms	16s 324 ms	10s 32 ms
200k	102 s 417 ms	95 s 184 ms	55-63s 411 ms	40s 233 ms
500k	229 s 259 ms	563 s 778 ms	379 ss 192ms	>245s 152 ms
1m	>10mins	>10mins	>10mins	>10 mins

Questions:

Do your own research and find which of the sequential version sorting algorithms lend themselves easily to parallel execution. If there are some that do not, explain why

Describe and justify the algorithm for parallelization you will be using.

Mergesort -
The way I approach this problem is to divide the array equally among the threads. So split the array up into subarrays using indices. A problem I ran into was that the size of the array does not divide by the number of array's evenly. My array can randomly generate numbers from 0-100. I added values of 101 to the end of the array until it was evenly divisible by the number of threads to make sure i have even subarrays and numbers were left off on the end. Once parallel sorting was done I removed the end numbers by a offset since 101 is guaranteed to be sorted to the end. To each thread I applied a variation of the sequential version of merge. Then I merged all the threads together. This method is good because all threads can continuously run without having to wait on another.

Bubblesort - I brought my thought process of splitting the array up into subarrays. I split the array into subarrays then I continuously did bubblesort on the arrays. I then compared the ends of the array to each other. I locked the higher array, to give time for the array to be resorted without having to worrying about doing another comparison on the end. When it finishes on a

section it unlocks it. This method made the most sense since numbers need to be compared at the ends. It's parallized but I don't think this is the most optimal method.

Which method performs best when implemented in parallel? Do your findings agree with what you would expect from analyzing the behavior of your program? Which method performs best sequentially? Justify your answers. Also, how does the performance of your parallel implementations change as you increase the number of threads? Why?

From my observations and research mergesort works best in parallel. Mergesort uses a divide and conquer method which lends itself very well to threading. Also each split doesn't interefere with one another so you don't have to make threads wait on locks.

The average case performance for mergesort is O(n log(n)), for bubblesort it's O(n^2) and for bucketsort it's O(n). Our results agree with this. In the sequential version mergesort outperformed bubblesort if we ignore the worse case scenarios.
For bubble sort I used mutex locks so threads have to wait on another. The way my code is implemented the more threads the more locks which will slow the code down. After a certain point adding more threads started increased time but the benefits decreased.
On 200000 units with 16 threads time was 36 secs, 32 threads time was 35 secs, with 64 it was 35 secs.
Each thread in mergesort works independently so the more threads the better. If we suppress the number of splits an array can do extra threads don't do any work but that is unlikely. More threads just speed up the process. After a certain point more threads actually slow down the time. I believe joining subarrays starts to outweights the benefits of the thread.

With 30 million units and 16 threads time was 2 s 386 ms, with 32 threads it was 3 s 512 ms, with 64 it was 5 s and 942 ms.

a) Which of the three algorithms when run sequentially has the best performance in order of complexity?
Based on average runtime bucketsort should have the best performance followed by mergesort.

b) Which of the three algorithms when run in parallel has the best performance in order of complexity? Why, how? Do you results agree with the analysis above? Why yes, why not?

Mergesort lends itself to parallelization the best since it's a dive and conquer algorithm. Based on our results mergesort run the fastest in parallel so it agrees with the theory