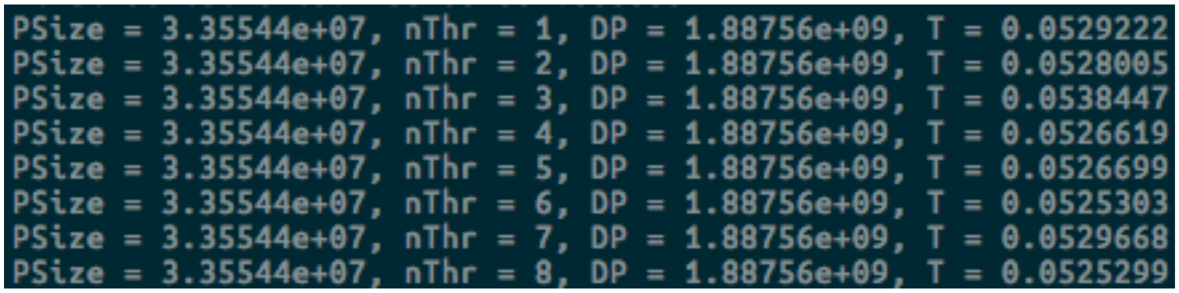
Brian Rodriguez
U0853593

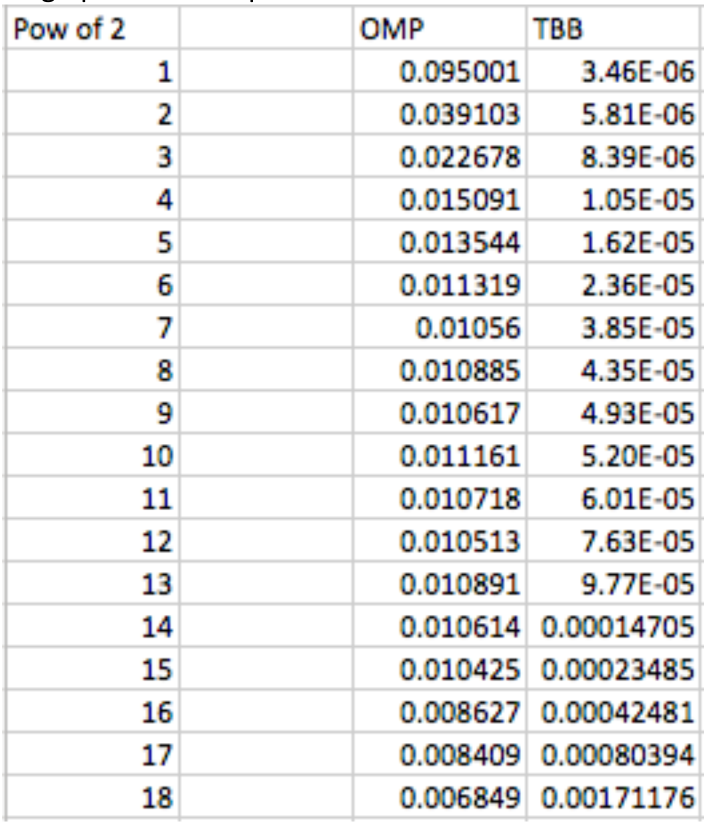
Assignment 3

1. My implementation of the parallel dot product gave me some troubles. The troubles I experienced were tbb automatically initializing the task scheduler to the max amount of hardware threads that my computer can handle. I have tried multiple implementations of the parallel dot product but to no avail. I even tried to go into the tbb source code to prevent the automatic default initializer from setting the number of threads equal to the number of default threads on my machine. The table that I have curated shows the parallel dot product running on 8 threads at all times so the times are relatively all the same.



```
PSize = 3.35544e+07, nThr = 1, DP = 1.88756e+09, T = 0.0529222
PSize = 3.35544e+07, nThr = 2, DP = 1.88756e+09, T = 0.0528005
PSize = 3.35544e+07, nThr = 3, DP = 1.88756e+09, T = 0.0538447
PSize = 3.35544e+07, nThr = 4, DP = 1.88756e+09, T = 0.0526619
PSize = 3.35544e+07, nThr = 5, DP = 1.88756e+09, T = 0.0526699
PSize = 3.35544e+07, nThr = 6, DP = 1.88756e+09, T = 0.0525303
PSize = 3.35544e+07, nThr = 7, DP = 1.88756e+09, T = 0.0529668
PSize = 3.35544e+07, nThr = 8, DP = 1.88756e+09, T = 0.0525299
```

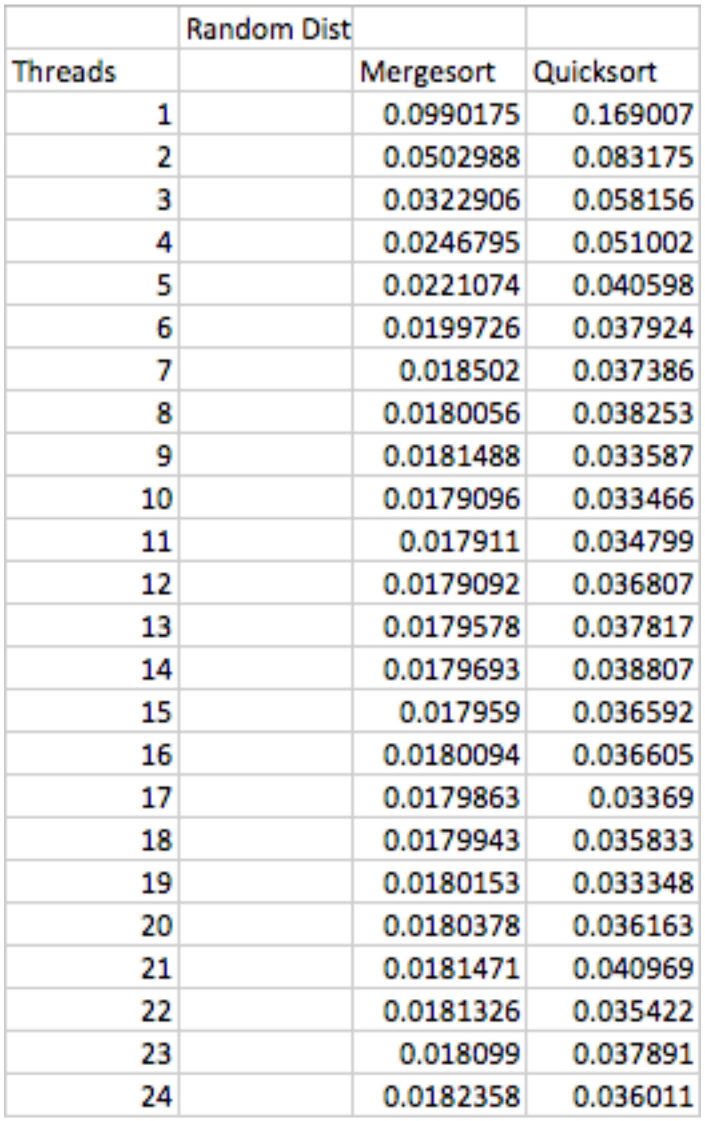
2. This is the time comparison for TBB and OMP. Both ran on 8 threads, the max for my machine and going up to 2 to the power of 20.



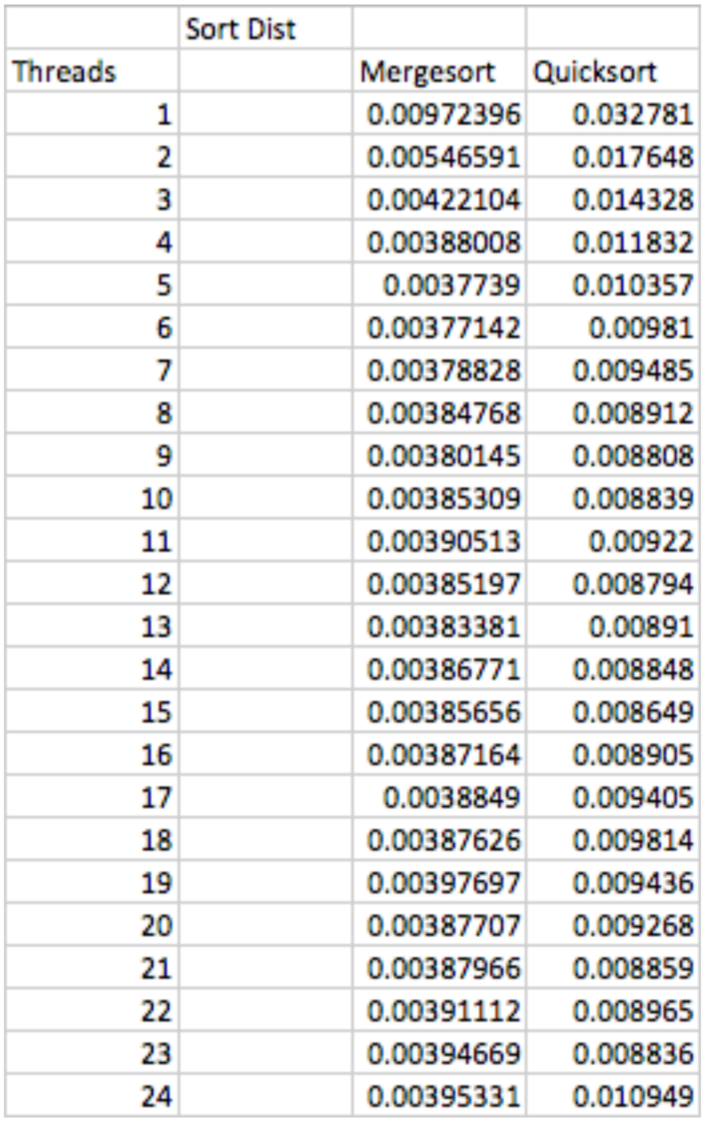
Pow of 2		OMP	TBB
1		0.095001	3.46E-06
2		0.039103	5.81E-06
3		0.022678	8.39E-06
4		0.015091	1.05E-05
5		0.013544	1.62E-05
6		0.011319	2.36E-05
7		0.01056	3.85E-05
8		0.010885	4.35E-05
9		0.010617	4.93E-05
10		0.011161	5.20E-05
11		0.010718	6.01E-05
12		0.010513	7.63E-05
13		0.010891	9.77E-05
14		0.010614	0.00014705
15		0.010425	0.00023485
16		0.008627	0.00042481
17		0.008409	0.00080394
18		0.006849	0.00171176

3. First and foremost, the call_parallel_merge_sort is the driver function that invokes the parallel merge sort of the two arrays that were passed in. It creates a new array which will be the result of the merge sort. The parallel merge sort function performs the act of breaking the arrays in pieces so that the threads can work on. It is a recursive function that splits the work into pieces and calls parallel invoke to recursively make threads. After a certain cut-off it will use the STL and call a stable sort, which is much more efficient for bigger datasets and saves the overhead of creating threads for erroneous work. Parallel merge, does a similar process as parallel merge sort but instead of sorting the arrays it is now merging four arrays into the final array. This method also has a cut off that uses the STL to merge the 4 arrays into the final array. Parallel merge spits the work up into chunks using the pointers as a means up dividing the work up so that the following calls to parallel marge can also work on the arrays.

4. The first table is a random distribution of numbers and the second is the sorted distribution.



	Random Dist		
Threads		Mergesort	Quicksort
1		0.0990175	0.169007
2		0.0502988	0.083175
3		0.0322906	0.058156
4		0.0246795	0.051002
5		0.0221074	0.040598
6		0.0199726	0.037924
7		0.018502	0.037386
8		0.0180056	0.038253
9		0.0181488	0.033587
10		0.0179096	0.033466
11		0.017911	0.034799
12		0.0179092	0.036807
13		0.0179578	0.037817
14		0.0179693	0.038807
15		0.017959	0.036592
16		0.0180094	0.036605
17		0.0179863	0.03369
18		0.0179943	0.035833
19		0.0180153	0.033348
20		0.0180378	0.036163
21		0.0181471	0.040969
22		0.0181326	0.035422
23		0.018099	0.037891
24		0.0182358	0.036011



	Sort Dist		
Threads		Mergesort	Quicksort
1		0.00972396	0.032781
2		0.00546591	0.017648
3		0.00422104	0.014328
4		0.00388008	0.011832
5		0.0037739	0.010357
6		0.00377142	0.00981
7		0.00378828	0.009485
8		0.00384768	0.008912
9		0.00380145	0.008808
10		0.00385309	0.008839
11		0.00390513	0.00922
12		0.00385197	0.008794
13		0.00383381	0.00891
14		0.00386771	0.008848
15		0.00385656	0.008649
16		0.00387164	0.008905
17		0.0038849	0.009405
18		0.00387626	0.009814
19		0.00397697	0.009436
20		0.00387707	0.009268
21		0.00387966	0.008859
22		0.00391112	0.008965
23		0.00394669	0.008836
24		0.00395331	0.010949

5.

a. The reason to employ a parallel merge is so that the developer of said algorithm can get rid of bottlenecks in the serial based algorithm. More work can be done by other threads since the problem can be broken into smaller chunks.

b. The work and span of parallel merge is nothing but the master theorem. The 2 is how many recursive calls that are made, the Θ(logn) is the extra work that is done for binary searching and T(N/2) is the division of problem size per recursive call. The span then is reduced into the geometric sequence Θ(log^2N). The work is a similar derivation but the overall work depends on a cut off factor in the algorithm. By design the quicksort implementation has 50% more comparisons but if we have a high cut off factor then the 50% comparisons become much smaller since we don't have as many comparisons to do work for.

c. The work and span of merge sort is nothing but the Master theorem. Since we have 2 recursive calls being made per mergesort, with N/2 the work being split every time and Θ(N) work that needs to be done every time then we get 2T(N/2) + Θ(N).

d. The span of both algorithms is the same since they both have the same amount of recursive calls per call to themselves and they both divide the work up into the same amount of chunks.

e. When you shrink the keys of a singly-threaded application the amount of work that is done is significantly less than if you had for example 32768 vs 16. The amount of comparisons is orders of magnitudes smaller than the latter.