Question 1

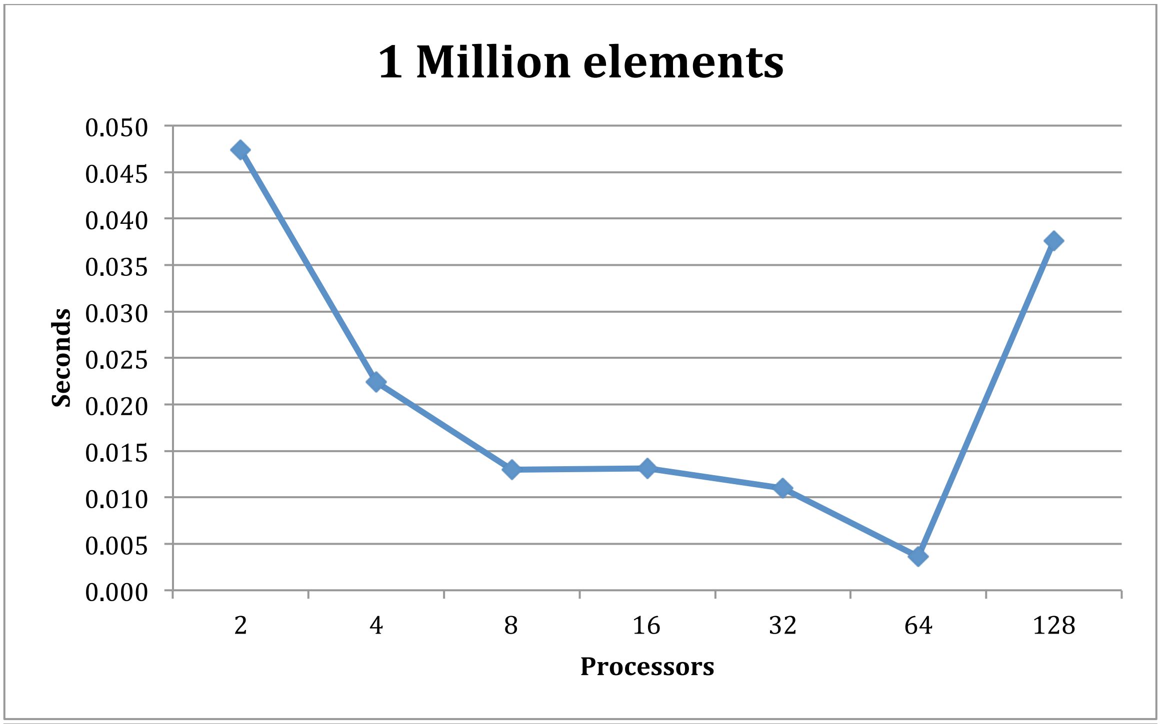
The algorithm consists of the details outlined in the project assignment. Initially, we sampled S elements from each local list. Subsequently, we pick a total of P - 1 pivots. These pivots are used to partition the dataset across the P processors. There were some minor adjustments made during step one to improve optimization. For example, on very small datasets, the formula for the number of sample elements we choose, S=12*P*$\log(N)$, will be large than the local array size. In this case, we pick the elements chosen to simply be half the length of the local list size. Subsequently, we must now communicate all of these selected elements back to processor zero so that the pivots can be selected. One method we tried to implement to improve the speed of communicating the selected elements was selecting the pivots within each process, and communicating those pivots back, but unfortunately, that does get weighed down by communication because there are cases where some processors would have selected more / less pivots than other processors. This can lead to some messy form of communicating data between processors.

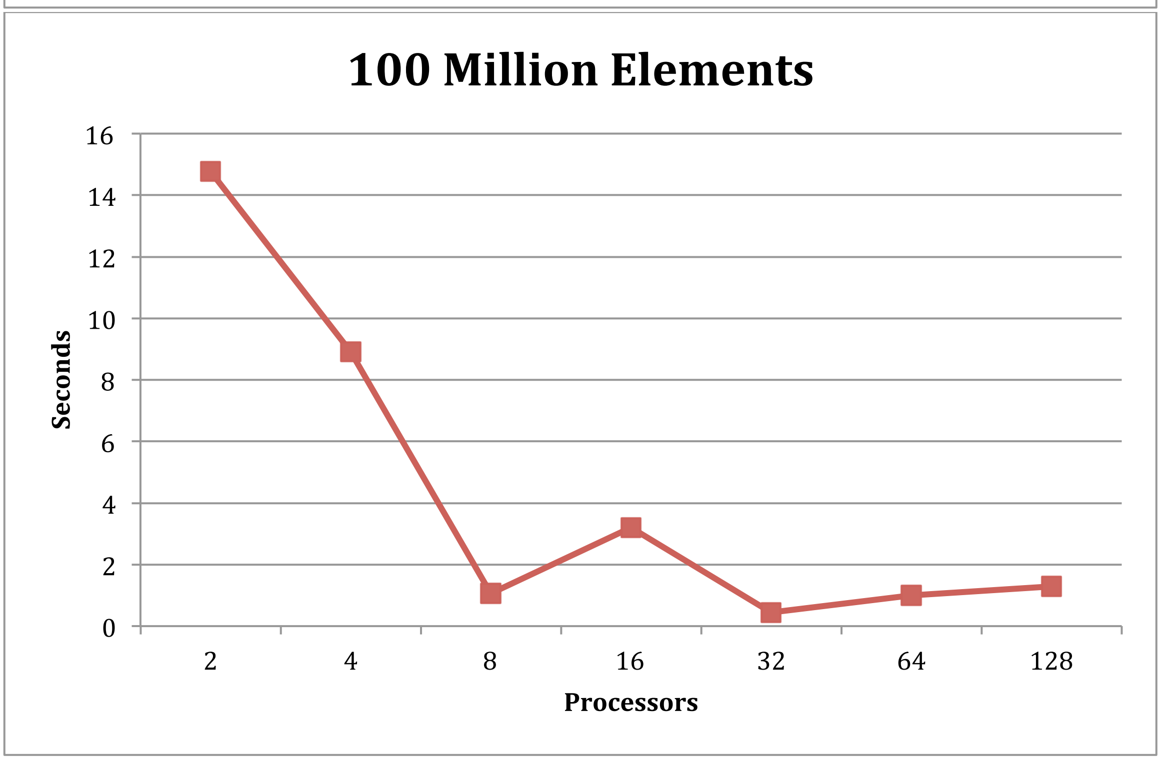
Afterward, we ran a sequential run through each local list for each processor respectively. This iteration was used to place each element in its bucket. This required two iterations, as we would have to calculate the number of elements in each bucket, which would later help us determine displacement. Both these values would be used for messaging between processors with MPIAlltoallv. Finally, we sent each bucket to the corresponding processor and performed a local sort on the data. The most strenuous parts of the program were understanding how exactly the Message Passing Interface could be exploited to communicating between processors. As we saw in the code, there were additional messages passed (like having an AlltoAll right before an AlltoAllv) in order to communicate receiving size before reading in arrays of different sizes.

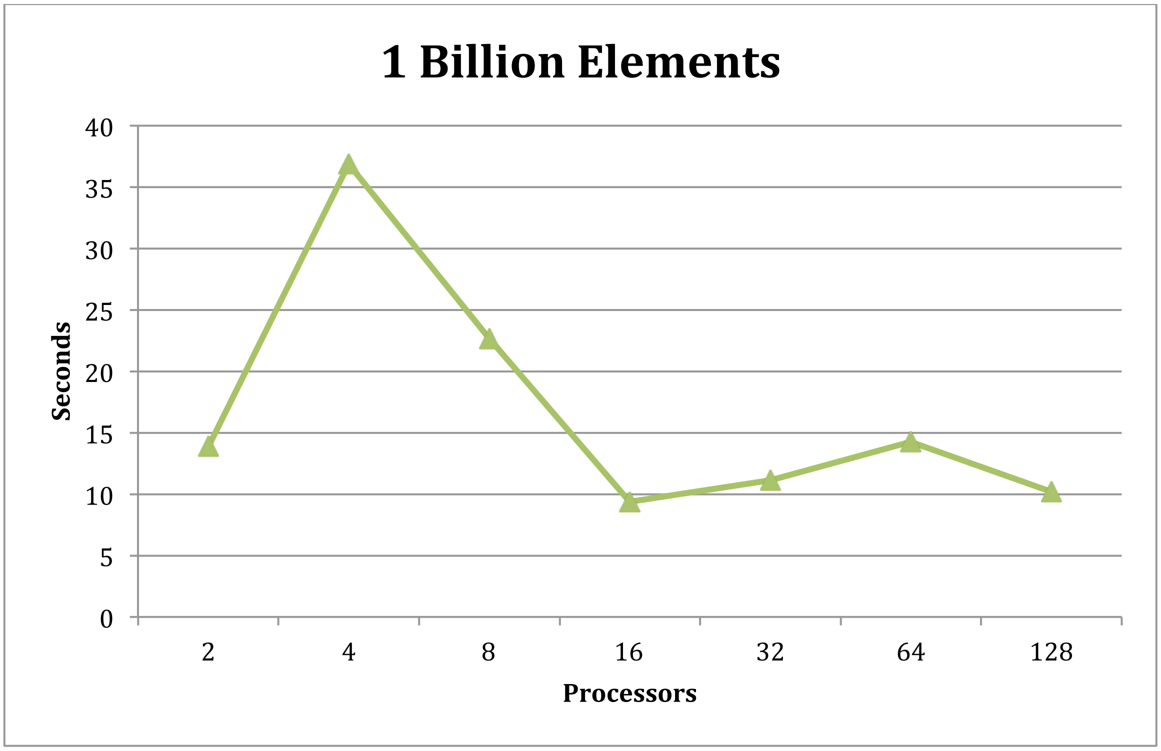
The 1 million-element array case, speedup for large amount of processors is still poor. This is mostly due to large communication overhead. This is indicated by the large amount of time taken to communicate in the Alltoall method. With smaller number of processors, the work there is relatively low, as fewer items need to be communicated between processes.

The reason a large number processors has poor performance is sometimes because of workload unbalancing. In the image below, I show the running time of each processor. As we can see, the workload is slightly unbalanced. We see a difference of 3 to 4 seconds in computation speed between the minimum and maximum thread. This is because certain pivots cause certain processors to have larger bucket values.

	1 Million elements	100 Million Elements	1 Billion Elements
2 Processors	took 0.0474s on 2 processors Speedup: 2.7279x	took 14.7649s on 2 processors Speedup: 1.1293x	took 13.8681s on 2 processors Speedup: 12.2352x
4 Processors	took 0.0224s on 4 processors Speedup: 5.7122x	took 8.9069s on 4 processors Speedup: 1.8764x	took 36.8845s on 4 processors Speedup: 4.6102x
8 Processors	took 0.0130s on 8 processors Speedup: 9.8777x	took 1.0518s on 8 processors Speedup: 16.0666x	Solution took 22.6257s on 8 processors Speedup: 7.7951x
16 Processors	took 0.0131s on 16 processors Speedup: 9.8391x	took 3.2057s on 16 processors Speedup: 5.3168x	took 9.3783s on 16 processors Speedup: 18.1339x
32 Processors	took 0.0110s on 32 processors Speedup: 13.3971x	took 0.4447s on 32 processors Speedup: 37.5835x	took 11.1450s on 32 processors Speedup: 15.7035x
64 Processors	took 0.0036s on 64 processors Speedup: 35.2309x	took 1.0076s on 64 processors Speedup: 16.8037x	took 14.2578s on 64 processors Speedup: 11.8803x
128 Processors	took 0.0376s on 128 processors Speedup: 7.7367x	took 1.2986s on 128 processors Speedup: 20.9423x	took 10.2177s on 128 processors Speedup: 21.3766x


1 Million elements
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Seconds
2
4
8
16
32
64
128
Processors


100 Million Elements
16
14
12
10
8
6
4
2
0
Seconds
2
4
8
16
32
64
128
Processors




```
Process 92 time is:                    [49145.828998531]
Process 78 time is:                    [49404.888680350]
Process 11 time is:                    [50980.860722368]
Process 67 time is:                    [49732.439555257]
Process 30 time is:                    [50645.104452822]
Process 94 time is:                    [49250.247147866]
Process 26 time is:                    [50728.557988245]
Process 14 time is:                    [50998.568465060]
Process 86 time is:                    [49408.678881271]
Process 99 time is:                    [48508.510527696]
Process 85 time is:                    [49462.580116262]
Process 58 time is:                    [50159.563697642]
Process 98 time is:                    [49281.433120806]
Process 43 time is:                    [50403.919609933]
Process 23 time is:                    [50861.003477155]
Process 18 time is:                    [50959.339673107]
Process 66 time is:                    [48607.611058978]
Process 101 time is:                   [49151.885942236]
Process 104 time is:                   [49059.533874330]
Process 82 time is:                    [49538.975977368]
Process 54 time is:                    [50239.321805944]
Process 39 time is:                    [50532.985180704]
Process 27 time is:                    [50840.479939157]
Process 117 time is:                   [48684.889857541]
Process 60 time is:                    [50104.579112987]
Process 113 time is:                   [48789.980288158]
Process 72 time is:                    [49868.414680037]
Process 63 time is:                    [50076.341600041]
Process 124 time is:                   [48638.093515037]
Process 52 time is:                    [50273.136103206]
Process 96 time is:                    [49392.830728873]
Process 127 time is:                   [48637.878893147]
Process 38 time is:                    [50601.797874348]
Process 32 time is:                    [50804.430829361]
Process 48 time is:                    [50411.107486143]
Process 59 time is:                    [48845.997068740]
Process 2 time is:                     [51757.781089196]
Process 5 time is:                     [51661.609203846]
Process 65 time is:                    [50150.751921465]
Process 116 time is:                   [48830.439634155]
Process 109 time is:                   [49085.580783547]
Process 88 time is:                    [49565.501340665]
Process 61 time is:                    [50208.191813930]
Process 20 time is:                    [51089.609177638]
Process 7 time is:                     [51562.265287270]
Process 68 time is:                    [50094.656684290]
Process 119 time is:                   [48879.787553567]
Process 12 time is:                    [51384.471552068]
```

In the image below, I examined all to all communication. Compared with the image before, we see that the all to all communication represents a fraction of the total overall cost. This indicates that communication is relatively fast.

```
Process 29 All to ALL communication:            [30.541392014] ms
Process 45 All to ALL communication:            [17.436667404] ms
Process 69 All to ALL communication:            [111.358966184] ms
Process 85 All to ALL communication:            [15.290939918] ms
Process 93 All to ALL communication:            [16.271824599] ms
Process 119 All to ALL communication:           [16.419962718] ms
Process 5 All to ALL communication:             [49.722810654] ms
Process 7 All to ALL communication:             [125.018955179] ms
Process 17 All to ALL communication:            [37.521429156] ms
Process 83 All to ALL communication:            [44.804427860] ms
Process 109 All to ALL communication:           [19.176331756] ms
Process 115 All to ALL communication:           [107.761102787] ms
Process 0 All to ALL communication:             [45.487122872] ms
Process 3 All to ALL communication:             [52.581243450] ms
Process 4 All to ALL communication:             [113.179619308] ms
Process 31 All to ALL communication:            [46.117832535] ms
Process 33 All to ALL communication:            [48.741617502] ms
Process 36 All to ALL communication:            [43.573080678] ms
Process 49 All to ALL communication:            [26.159727975] ms
Process 81 All to ALL communication:            [47.809987213] ms
Process 12 All to ALL communication:            [136.748204153] ms
Process 18 All to ALL communication:            [54.241272795] ms
Process 19 All to ALL communication:            [124.815627380] ms
Process 20 All to ALL communication:            [61.460765253] ms
Process 22 All to ALL communication:            [80.094103585] ms
Process 24 All to ALL communication:            [150.880065019] ms
Process 25 All to ALL communication:            [79.090357962] ms
Process 26 All to ALL communication:            [118.676269893] ms
Process 28 All to ALL communication:            [126.618180162] ms
Process 32 All to ALL communication:            [46.546941274] ms
Process 34 All to ALL communication:            [40.950956929] ms
Process 35 All to ALL communication:            [48.471072834] ms
Process 37 All to ALL communication:            [65.606050834] ms
Process 40 All to ALL communication:            [66.043892410] ms
Process 41 All to ALL communication:            [61.028520577] ms
Process 42 All to ALL communication:            [38.434791320] ms
Process 44 All to ALL communication:            [45.925678773] ms
Process 48 All to ALL communication:            [129.378049838] ms
Process 50 All to ALL communication:            [69.198433659] ms
Process 52 All to ALL communication:            [142.512796971] ms
Process 56 All to ALL communication:            [65.696294623] ms
Process 57 All to ALL communication:            [58.367489633] ms
Process 60 All to ALL communication:            [62.983809010] ms
Process 64 All to ALL communication:            [41.320492252] ms
Process 65 All to ALL communication:            [41.457657877] ms
Process 66 All to ALL communication:            [72.185021767] ms
Process 68 All to ALL communication:            [40.786163183] ms
Process 72 All to ALL communication:            [38.032625394] ms
Process 74 All to ALL communication:            [37.290731940] ms
Process 76 All to ALL communication:            [36.244424788] ms
Process 80 All to ALL communication:            [126.352209540] ms
```

Breadth First Search

Optimization:

1. Parallelize top-down BFS:

   Using OpenMP to parallelize sol->distances array initialization:

For each round of frontier iteration, since the computation is independent, use OpenMP to parallelize. The process to build up new frontier has multiple threads updating the shared array, we use __sync_bool_compare_and_swap to make sure the writes are correct.

2. Optimize the compare_and_swap operation:

Instead of calling __sync_bool_compare_and_swap for every write operation, we call __sync_bool_compare_and_swap on behalf of a bunch of write operations. Specifically, in top-down method, for every node in frontier we only call __sync_bool_compare_and_swap once to reserve the right amount of slots on shared new_frontier after collection all the child nodes of current node, then subsequent write operations do not need call __sync_bool_compare_and_swap. And in bottom-up method, instead of assigning each node a thread, we group bunches of nodes together then assign one thread to the group, setting the group size to 1000. And for each group it only needs one __sync_bool_compare_and_swap call to reserve the right amount of slots on shared new_frontier, then subsequent write operations do not need call __sync_bool_compare_and_swap. In both methods, it reduces the number of atomic __sync_bool_compare_and_swap function calls a lot and thus reduce the waiting time in the spinning while loop dramatically.

3. Minor optimization:

The program expand the frontier in a layer-by-layer manner, and since every edge has the same weight of 1, so instead of reading parent node's accumulative weight, we pass the calculated distance into step functions and thus save lots of memory read accesses.

Bottom-up method implementation notes:

To represent the frontier, we use the same data structure as the one in top-down method; we do that to make the implementation of hybrid method easier. Since every edge has the same weight of one, and we pass the current accumulative distance in the variable of step, so for all the incoming nodes of each node, if the distance of incoming node is step-1, we know it is in the current frontier, and then set the current node's distance to step and add it to the new frontier later.

Algorithm of hybrid method:

The hybrid method combines the advantages of bottom-up method and top-down method. Top-down method has better performance when frontier is small, and bottom-up method has better performance when frontier is large. So we use the ratio between graph's total nodes number and current frontier's size, if the ration is less than 10, it indicates the frontier is too large for top-down method and otherwise it is too small for bottom-up method. Since in our implementation, the bottom-up and top-down share the same data structure of frontier, it is seamless to switch between the two methods.

PART TWO:

Performance analysis:

1. Where is the synchronization in each your solutions? Do you do anything to limit the overhead of synchronization?

We use the __sync_bool_compare_and_swap atomic function call to synchronize the write operations on shared new_frontier among all the running threads. Instead of doing the CAS on every write operation, we group the write operations and call CAS once on behalf of the group write operations to reserve right amount of slots on the shared new_frontier struct.

2. Why do you think your code (and the TA's code) is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)

We think the problem lies on shared data access synchronization. While updating new_frontier, the write operations cannot be totally parallelized, and basically updating the new_frontier->count needs to be serialized to yield correct results. So the nature of the program, i.e. sequential part of the program, limited the maximum parallel speedup.

3. When you run your code on Blacklight with more than 16 threads, do you see a noticeable drop off in performance? Why do you think this might be the case?

Yes, we notice a noticeable performance drop off for all methods, which is the same as the outcome of reference result. We think the problem is that, although adding more threads results in more work are processing in parallel, but the contend overhead of write to shared new_frontier is also increasing, and when threads number is too large, the benefit of parallel is overcome by the overhead of contending to update the shared new_frontier.

BFS Performance for random_50m.graph:

Gates 3000:

Num Threads	Sequence Top Down	Top Down	Bottom Up	Hyrbid
1	11.421	13.211(1x)	76.060(1x)	8.838(1x)
2	11.441	10.040(1.32x)	33.574(2.27x)	4.694(1.88x)
4	11.426	6.043(2.19x)	23.167(3.28x)	2.940(3.00x)
6	11.322	5.250(2.52x)	20.805(3.66x)	2.366(3.74x)
8	11.845	5.559(2.38x)	22.791(3.34x)	2.723(3.25x)
12	18.362	4.822(2.74x)	19.850(3.83x)	2.434(3.63x)

Blacklight:

Num Threads	Sequence Top Down	Top Down	Bottom Up	Hyrbid

1	35.915	38.932(1x)	105.835(1x)	13.957(1x)
2	35.743	24.496(1.59x)	53.549(1.98x)	7.475(1.88x)
4	37.728	16.719(2.33x)	27.382(3.87x)	3.948(3.54x)
8	36.370	11.663(3.34x)	14.588(7.25x)	2.149(6.49x)
16	35.823	16.769(2.32x)	14.113(7.50x)	1.919(7.27x)
32	37.194	68.648(0.57x)	33.448(3.16x)	5.463(2.55x)
64	35.796	122.761(0.32x)	19.353(5.47x)	6.353(2.20x)
128	35.688	111.368(0.35x)	11.980(8.83x)	8.396(1.66x)

BFS Performance for random_1m.graph:

Gates 3000:

Num Threads	Sequence Top Down	Top Down	Bottom Up	Hyrbid
1	0.181	0.195(1x)	0.270(1x)	0.052(1x)
2	0.201	0.109(1.79x)	0.137(1.97x)	0.029(1.79x)
3	0.235	0.089(2.19x)	0.094(2.87x)	0.022(2.36x)
4	0.141	0.073(2.67x)	0.080(3.36x)	0.022(2.36x)
5	0.143	0.060(3.25x)	0.075(3.60x)	0.018(2.89x)
6	0.142	0.070(2.79x)	0.073(3.70x)	0.017(3.06x)
12	0.158	0.045(4.33x)	0.076(3.55x)	0.015(3.45x)