15-440 Distributed Systems
4 May 2013

Boya Yang (boyay)
Shikun Zhang (shikunz)

Clustering Data Points and DNA Strands Using MPI

Parallel Algorithm

Overview

When running parallel k-means algorithm, we divide all data into several chunks, and each processor only computes data in one chunk. All processors will share one list of centroids, which in each iteration is updated/recalculated by the root node(processor with rank 0) after it collects data from each processor. We iterate until the centroids converge.

Procedure

1. Initialize and distribute data
   Node with rank zero (the root) loads and partitions the data into *n* chunks, where n is the number of processors. By calling MPI.scatter, the root distributes different data to corresponding processors. Also, the root picks the initial positions of the centroids, and the broadcasts it to all nodes.

2. Calculate and flatten membership
   Each node receives and processes its chunk of data, assigning each data point to which cluster it belongs using the broadcasted list of centroids. (we will call this the assignment of membership of data points.) The root (rank 0) uses MPI.gather to collect all membership information from each node, and then flattens the list of lists into one list.

3. Recalculate and Sync centroids
   The root, after gathering all membership info, uses that data to calculate a new list of centroids, and then broadcasts it to all participating nodes.

4. Repeat Step 2 until convergence
   We compare the old list with the new list of centroids, if the difference between the two is significant, we go back to Step 2 and repeat, otherwise we can exit the loop, and then write the new centroids into a file.

PseudoCode

```
fun main(k) {
    if (rank == 0) {
        //Step1: root loads and partitions date
        data = loadDataFromFile(f, ‘r’’)
        chunk = a list of split chunks of data
        newCentroids = getInitialCentroids(k)
    } else {
        chunk = None
        newCentroids = None
    }
    newCentroids = MPI.broadcast(newCentroids, root=0)
    chunk = MPI.scatter(chunk, root=0)
    //Step 4
    while (not converging(newCentroids, oldCentroids)) {
        oldCentroids = newCentroids
        //Step 2
        membershipInfo = assignMembership(chunk, newCentroids)
        // root collects data from every node
        allMemberInfo = MPI.gather(membershipInfo, root=0)
        if (rank == 0)
            //Step 3
            newCentroids = updateCentroids(allMemberInfo)
        newCentroids = MPI.broadcast(newCentroids, root=0)
    }
    writeToFile(newCentroids)
    return;
}
```

Experimentation and Analysis

Data Generation

Inputs used for the analysis was generated using the PointGenerator and DNAGenerator that we wrote. All inputs were generated with a p-value (number of points per cluster) of 10000, because we feel that testing our program on a large dataset would be a better reflection of data intensive programming in the real world. We generated inputs that used k-values (number of clusters) ranging from 2 to 10, for both point input and DNA input.

Method

We used Python's cProfile package to time our processes. We first ran the sequential kMeans program. Each of the 20 inputs (one for each value of k) was ran 5 times and an average time taken obtained for analysis. The parallel kMeans program was run in the same way, but with the entire process repeated for n-values (number of processes) 2, 4, 8 and 12. Also, when recording the time taken for each input, we used the time taken by the slowest processor. All tests were ran on ghc10, which has 4 cpu's.

Results *(Full Results attached at the end)*

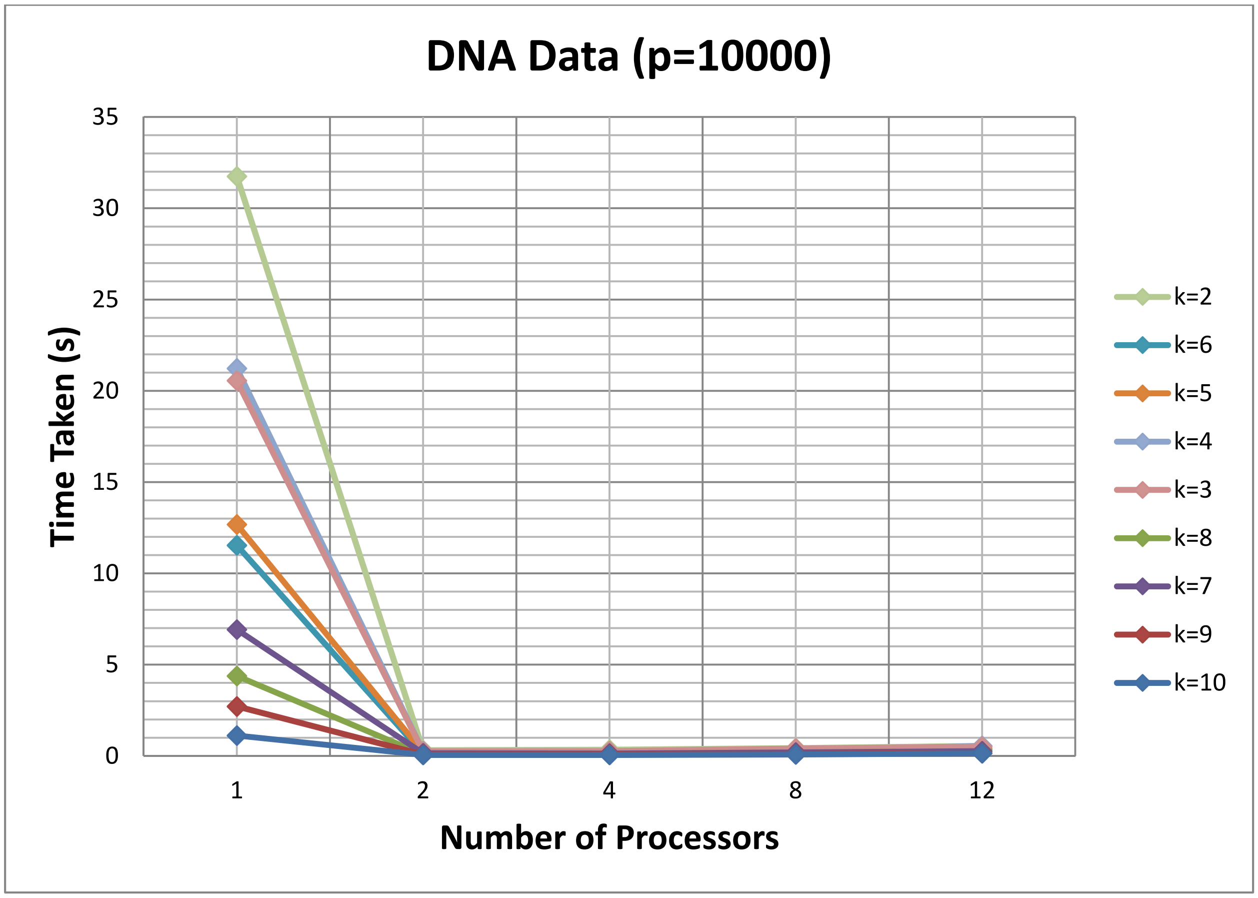
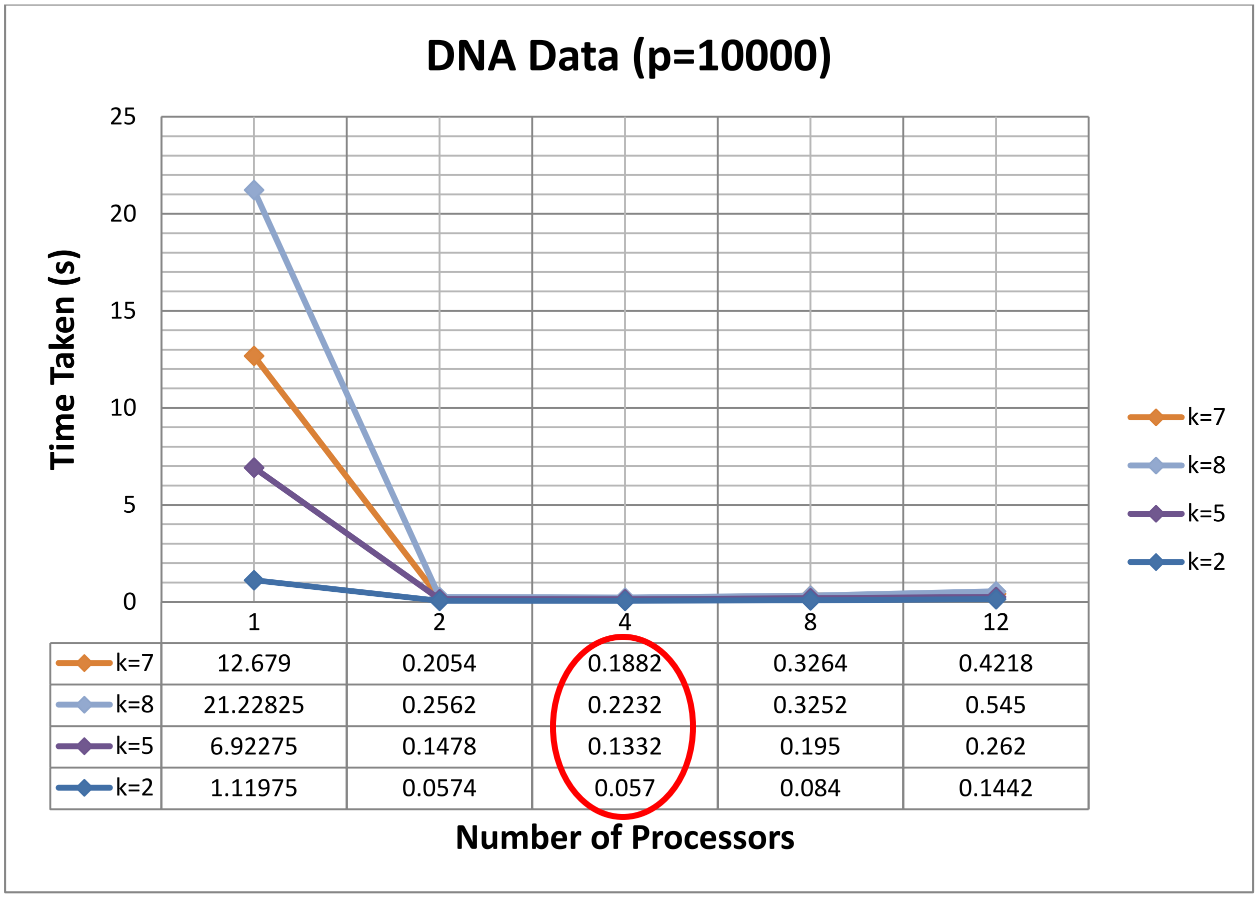


Figure1: Graph of time against number of processors for DNA data

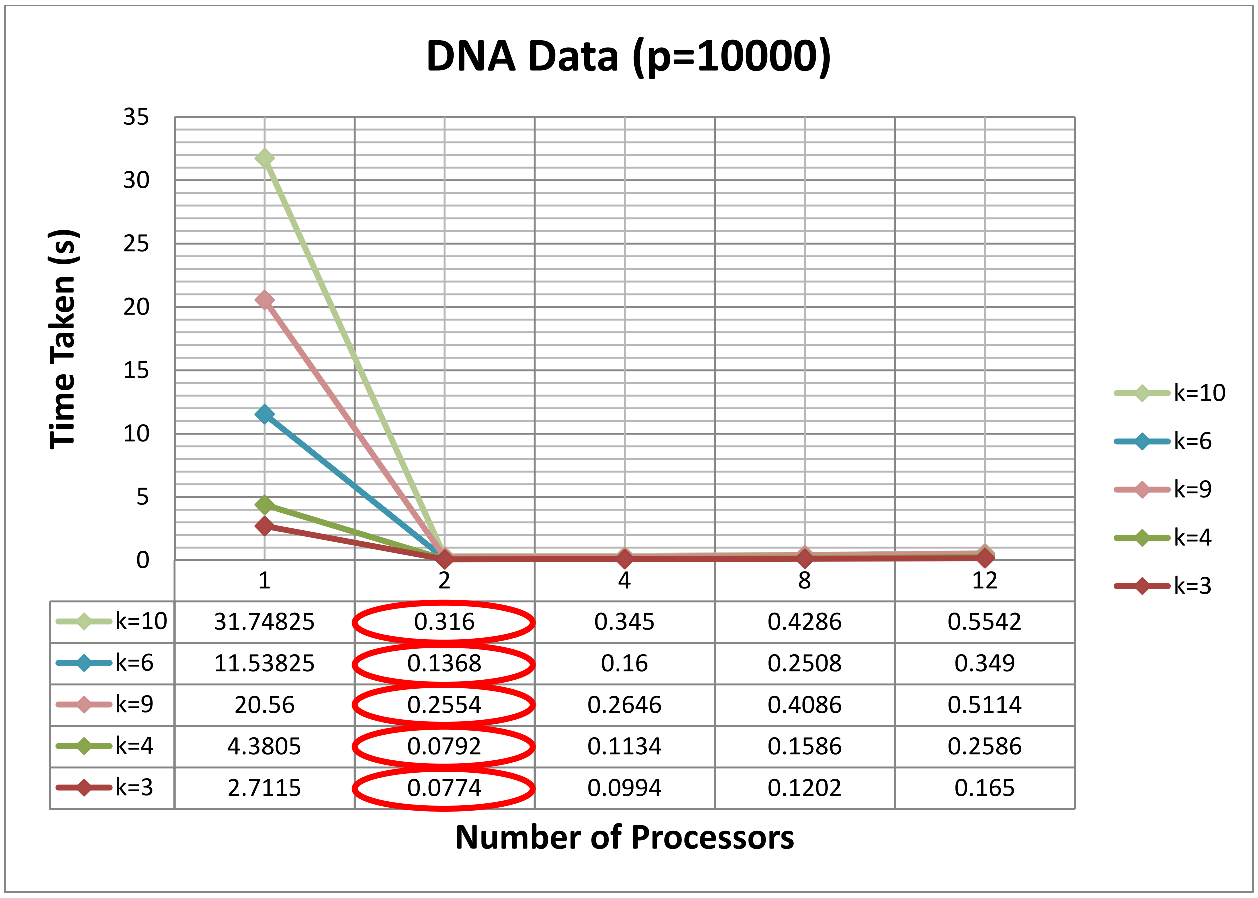
From the results that we obtained as shown in Figure 1, we can see that parallelizing our algorithm has drastically reduced the time taken to run the kMeans algorithm. The effect is more pronounced with higher k (more clusters) because with a fixed p-value of 10000, the total number of points increases with greater k. Since the parallel algorithm assigns each processor to process a chunk of points concurrently, it is much more efficient than the sequential algorithm, in which all points have to wait to be processed by a single processor. With more points, there is also a greater potential for parallelization, and hence the parallel algorithm is more useful on larger inputs.



	1	2	4	8	12
k=7	12.679	0.2054	0.1882	0.3264	0.4218
k=8	21.22825	0.2562	0.2232	0.3252	0.545
k=5	6.92275	0.1478	0.1332	0.195	0.262
k=2	1.11975	0.0574	0.057	0.084	0.1442

Figure 2: Graph of time against number of processors for DNA data for k = 2, 5, 7, 8

Taking a closer look at the results, we can see from Figure 2 that for k-values of 2, 5, 7, 8, the most efficient number of processors to use is 4. Note that we ran all our tests on the ghc10 machine, which has 4 cpu's. This is an expected result as using less than 4 processors would mean that we were not making full use of all the processor power available, and using more than 4 processors on a machine with only 4 cpu's would result in high overhead of communications between processors without the added efficiency. As the number of processors is increased beyond 4, we see a gradually increasing trend in the time taken for the program to run. This is because the cost of the increased overhead has outweighed the added benefit of greater parallelization.



	1	2	4	8	12
k=10	31.74825	0.316	0.345	0.4286	0.5542
k=6	11.53825	0.1368	0.16	0.2508	0.349
k=9	20.56	0.2554	0.2646	0.4086	0.5114
k=4	4.3805	0.0792	0.1134	0.1586	0.2586
k=3	2.7115	0.0774	0.0994	0.1202	0.165

Figure 3: Graph of time against number of processors for DNA data for k = 3, 4, 6, 9, 10

However, Figure shows contradictory evidence that the most efficient number of processors to use is 2. This can be explained by the fact that the tests were run on a public machine that is used my multiple users at the same time. We checked that there was one other user who was sharing the ghc10 machine with us when our tests were running. The fact that our results show the 'sweet spot' to be 2 for some values of k indicated that we were only making full use of 2 of the total 4 cpu's on the machine when these tests were running. To obtain more accurate results, we can run our tests on a personal machine.

Full Results

	sequential					
k	1	2	3	4	5	average
2	0.301	0.304	0.306	0.307	0.293	0.3022
3	3.138	3.698	4.24	1.499	1.708	2.8566
4	11.595	10.517	10.817	2.853	2.975	7.7514
5	15.791	20.504	12.379	9.818	22.932	16.2848
6	11.92	11.413	31.367	12.582	8.894	15.2352
7	34.775	27.808	83.642	59.828	57.128	52.6362
8	64.653	88.934	60.707	19.548	87.001	64.1686
9	69.785	23.142	51.978	38.526	56.11	47.9082

	sequential					
k	1	2	3	4	5	average
2	1.118	1.126	1.117	1.118	1.13	1.11975
3	2.185	3.215	2.176	3.27	2.176	2.7115
4	5.402	3.493	5.182	3.445	7.198	4.3805
5	7.711	9.67	5.159	5.151	7.242	6.92275
6	10.189	10.957	11.27	13.737	10.708	11.53825
7	14.502	13.053	9.498	13.663	14.173	12.679
8	24.211	17.015	17.463	26.224	18.199	21.22825
9	15.741	22.771	22.626	21.102	36.722	20.56
10	28.144	45.766	25.955	27.128	27.725	31.74825

k	2 processors					
2	1	2	3	4	5	average
3	0.35	0.411	0.707	0.408	0.442	0.4636
4	1.704	1.873	1.709	2.008	2.188	1.8964
5	4.531	3.568	15.058	11.756	14.532	9.889
6	18.01	7.714	4.523	7.416	5.593	8.6512
7	6.362	37.742	8.037	8.239	14.173	14.9106
8	89.513	32.043	77.398	56.343	79.274	66.9142
9	102.557	8.937	68.137	10.169	80.288	54.0176
	128.909	84.615	137.269	141.128	91.897	116.7636

k	2 processors					
2	1	2	3	4	5	average
3	0.056	0.057	0.061	0.058	0.055	0.0574
4	0.084	0.079	0.091	0.048	0.085	0.0774
5	0.063	0.084	0.066	0.063	0.12	0.0792
6	0.098	0.157	0.168	0.167	0.149	0.1478
7	0.185	0.175	0.099	0.09	0.135	0.1368
8	0.207	0.207	0.196	0.208	0.209	0.2054
9	0.282	0.252	0.244	0.256	0.247	0.2562
10	0.292	0.294	0.171	0.303	0.217	0.2554
	0.328	0.316	0.299	0.317	0.32	0.316

4 processors

k	1	2	3	4	5	average
2	0.329	0.306	0.305	0.306	0.303	0.3098
3	1.465	1.517	1.477	3.776	1.3	1.907
4	2.467	11.588	2.244	13.859	2.57	6.5456
5	3.002	13.026	14.221	13.609	27.158	14.2032
6	4.245	13.574	47.066	34.53	10.912	22.0654
7	22.321	70.561	16.091	30.899	6.58	29.2904
8	70.182	39.096	45.67	44.863	37.677	47.4976
9	197.668	58.613	44.891	59.157	59.934	84.0526

4 processors

k	1	2	3	4	5	average
2	0.053	0.078	0.052	0.051	0.051	0.057
3	0.131	0.079	0.128	0.08	0.079	0.0994
4	0.129	0.116	0.108	0.107	0.107	0.1134
5	0.134	0.131	0.132	0.136	0.133	0.1332
6	0.156	0.156	0.158	0.16	0.17	0.16
7	0.188	0.187	0.184	0.191	0.191	0.1882
8	0.239	0.219	0.213	0.217	0.228	0.2232
9	0.258	0.266	0.274	0.259	0.266	0.2646
10	0.462	0.313	0.316	0.316	0.318	0.345

8 processors

k	1	2	3	4	5	average
2	0.367	0.373	0.473	0.575	0.468	0.4512
3	1.632	1.627	1.64	5.758	1.991	2.5296
4	2.087	2.321	11.169	3.074	2.079	4.146
5	3.232	12.861	14.143	9.379	15.324	10.9878
6	12.517	20.185	13.501	50.289	5.655	20.4294
7	11.449	59.827	67.096	46.682	33.514	43.7136
8	17.549	19.247	46.43	91.58	99.369	54.835
9	24.364	12.244	73.004	12.778	72.698	39.0176

8 processors

k	1	2	3	4	5	average
2	0.076	0.091	0.071	0.1	0.082	0.084
3	0.124	0.111	0.122	0.111	0.133	0.1202
4	0.157	0.163	0.164	0.153	0.156	0.1586
5	0.202	0.199	0.202	0.19	0.182	0.195
6	0.302	0.238	0.229	0.247	0.238	0.2508
7	0.357	0.384	0.29	0.233	0.368	0.3264
8	0.304	0.316	0.317	0.373	0.316	0.3252
9	0.467	0.456	0.369	0.374	0.377	0.4086
10	0.423	0.536	0.396	0.395	0.393	0.4286

12 processors

k	1	2	3	4	5	average
2	0.458	0.542	0.496	0.593	0.587	0.5352
3	1.88	6.334	6.554	3.637	1.571	3.9952
4	2.212	2.441	3.302	18.218	3.372	5.909
5	3.693	28.61	38.697	8.98	12.876	18.5712
6	6.625	9.397	21.653	5.683	20.766	12.8248
7	7.318	33.012	5.41	24.809	48.481	23.806
8	42.645	25.591	73.235	36.615	81.546	51.9264
9	60.582	76.206	72.604	24.717	12.244	49.2706

12 processors

k	1	2	3	4	5	average
2	0.166	0.23	0.092	0.126	0.107	0.1442
3	0.107	0.189	0.199	0.16	0.17	0.165
4	0.224	0.258	0.309	0.275	0.227	0.2586
5	0.214	0.258	0.265	0.21	0.363	0.262
6	0.435	0.329	0.408	0.235	0.338	0.349
7	0.394	0.38	0.516	0.415	0.404	0.4218
8	0.555	0.533	0.46	0.423	0.754	0.545
9	0.48	0.499	0.592	0.523	0.463	0.5114
10	0.564	0.549	0.566	0.558	0.534	0.5542