CT-LSD Assignment 2

Albin Jansfelt, Amanda Allgurén, Lukas Martinsson

March 2022

1 Problem 1

The boss wants to have the summary statistics done every morning by 8:00. Since the analysis workflow starts at 20:00 the previous day, the analysis is limited to a maximum of 12 hours. The two main aspects when comparing the running time of physical servers vs cloud based is the reading and writing (I/O) the data and the computational time itself.

The first aspect to discuss is what the issue is why the analysis is taking so long time. What are the bottlenecks in this procedure? If the process is CPU-bound it means that the main bottlenecks are the computations done in the analysis. If we assume that the code can be further parallelized, and will by that reduce the time for analysis, the number of cores is of interest. To reach the 8:00 goal, a speedup of a minimum $20/12 \approx 1.67$ is needed. However, this is true if the proportion of code which can be parallelized is close to 1. This could be analyzed with Amdahl's law to see what the total speedup would be if parallelizing some of the computations.

The process could also be I/O-bound, meaning that the main bottleneck is reading and writing data. In theory, the cloud has unlimited computational power (since it can be scaled to meet the needs). The issue will instead lay in if it is possible to send and receive the data amount (in this case 1,3 TB) to the cloud within the time limit. This will be affected by the bandwidth, which will contribute to a cost itself varying with the capacity. This cost will have be compared to the cost of having on site servers. Additional aspects of this is if the increased complexity is only temporary or will continue to increase. If the spike is temporary, even a higher cost of bandwidth could be preferrable since the capacity of this can be adjusted over time more easily.

2 Problem 2

Figure 1 shows the speedup for the pi approximation for different number of cores used. For this, the accuracy sought was 0.00001 and the number of sampling done for each worker each turn was 1000. We do know that the speedup doesn't look as expecte and the problem is probably due to threads not being closed correctly.

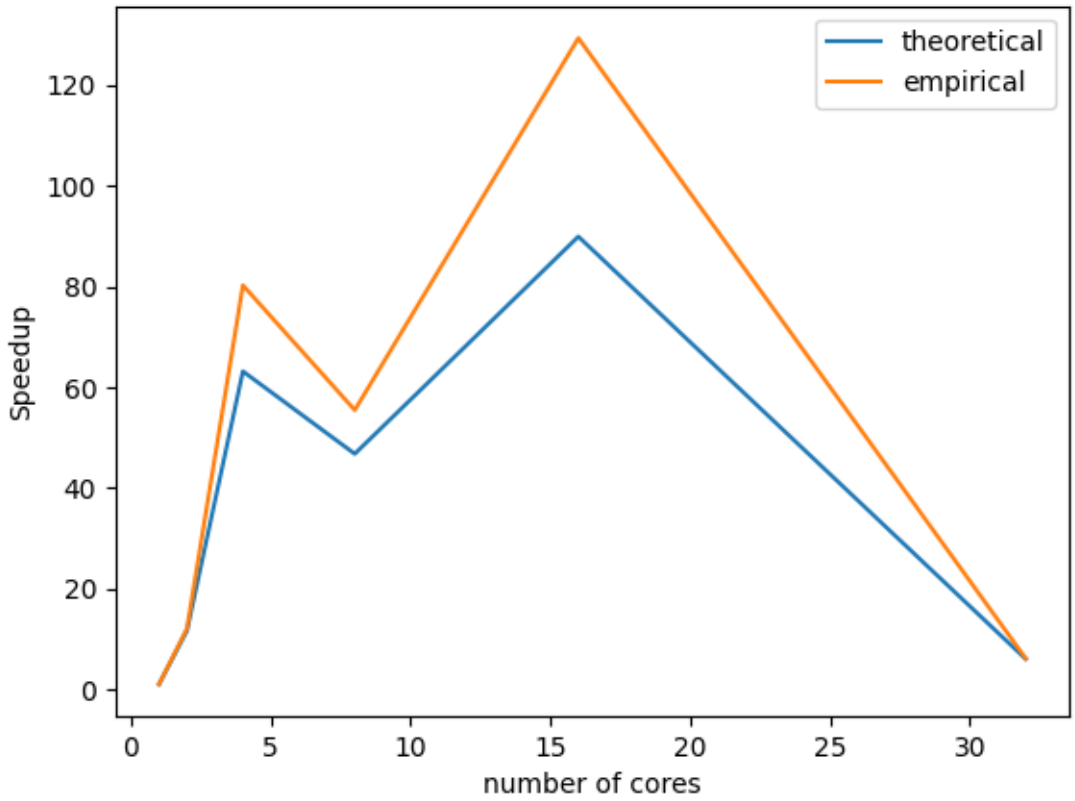

Figure 1: Caption

3 Problem 3

3.1 (a)

Two main blocks in kmeans function are found meaningful to parallelize, where each datapoint is to be considered by the program.

One being row 48 to 52. In this section, each datapoint is being assigned to its closests centroid, a count of number of datapoints assigned to each cluster kept and variation computed. If each of the workers keep a seperate list of this, the information can after the parallelization being merged, Cluster size and variation of each cluster will be a list of number of clusters and can thereby be aggregated after the parallelization is complete. The c[i] lists of each data point can be concatenated into one single list, alternatively be updated as one list by all workers since the same index will not be edited more than once each iteration.

The other section which can be parallelized is row 60 to 61. In this section, the centroids are recomputed by taking the mean of the datapoints of each cluster. This can be parallelized over the different workers, each keeping a list of the cluster counts, and then aggregated together and divided by their cluster sizes.

What can be seen as a sequential bottleneck is the step in between these two identified sections. No worker can start recompute the centroids before all data points have been assigned, since the centroids are dependent on all data points. This means that all workers have to wait for the others to finish before entering the next section.

In computeClustering, the code also needs to be sequentially run, for example plotting the final result.

Figure 2 and 3 shows some ideas of how to implement this parallelization in the code, by using starmap to map the new functions findCentroids and countClusters to each separate worker and then aggregate or append the results received by each worker.

3.2 (b)

The kmeans program was run with 10000 datapoints. The total time taken was 29.57 [s]. There are 2 section which we intend to parallelize, and are described above. For the first section, the time taken was 27.144 [s]. For the second section, the time taken was 2.417 [s]. The sum of the two parallelizeable sections are $27.144 + 2.417 = 29.561$ [s], which means that there are only $29.570 - 29.561 = 0.009$ [s] that are serialized.

3.3 (c)

From Amdahl's law we have

$$S_{total}(f, S) = \frac{1}{1 - f + f/S} \tag{1}$$

```
        variation = np.zeros(k)
        cluster_sizes = np.zeros(k, dtype=int)

        '''
        splitted_data = np.array_split(data, workers)
        p = multiprocessing.Pool(workers)
        c_w, cluster_size_w, variation_w = p.starmap(findCentroids, (splitted_data, centroids))
        cluster_size <- aggregate cluster_size_w of each worker to a single list
        variation <- aggregate variation_w of each worker to a single list
        c <- append each c_w to list which shows cluster index of each data point
        '''

        delta_variation = -total_variation
        total_variation = sum(variation)
        delta_variation += total_variation
        logging.info("%3d\t\t%f\t%f" % (j, total_variation, delta_variation))

        # Recompute centroids
        centroids = np.zeros((k,2)) # This fixes the dimension to 2

        '''
        centroids_w = p.starmap(countClusters, (splitted_data, centroids, c))
        centroids <- aggregate centroids_w of each worker to a single list
        '''

        centroids = centroids / cluster_sizes.reshape(-1,1)
```

Figure 2: Functions called by starmap to parallelize parts of code

```
'''
def findCentroids(tup):
    data = tup[0]
    centroids = tup[1]

    N = len(data)
    c = np.zeros(N)
    cluster_sizes = np.zeros(len(centroids))
    variation = np.zeros(len(centroids))
    for i in range(N):
        cluster, dist = nearestCentroid(data[i],centroids)
        c[i] = cluster
        cluster_sizes[cluster] += 1
        variation[cluster] += dist**2
    return c, cluster_sizes, variation

def countClusters(tup):
    data = tup[0]
    centroids = tup[1]
    c = tup[2]
    N = len(data)
    for i in range(N):
        centroids[c[i]] += data[i]
    return centroids
'''
```

Figure 3: Functions created to be run in parallel

in our case $f = 29.561/29.570 = 0.9997$. Further, we assume that if we use W workers the parallel part will take $29.561/W$ seconds. Thus, 4 workers gives a speedup of 4, i.e. $S = 4$ and 8 workers gives a speedup of 8, $S = 8$.

Hence, $S_{total}(0.9997, 4) = 3.9964$ [s] and $S_{total}(0.9997, 8) = 7.9832$ [s]. As expected, since f is very close to 1, the theoretical total speedup is very close to the speedup, S.