DIT065 - Computational Techniques for Large-scale Data Assignment 2

Huw Fryer
Cem Mert Dalli
Stefano Ribes

April 3, 2022

1 Problem 1

Since we have limited information on the analysis workflow at the factory, we assume that most of the tasks can benefit from parallelization and a significant acceleration can be achieved. As underlined in Lecture 2 of the course (Schliep, 2022), the theoretical speedup that can be achieved through parallelization is bounded by Amdahl's law:

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

where S refers to the percentage of running time that can be parallelized by a factor of S. With this knowledge, we are making a decision between purchasing centrally-located high performance computing servers (HPC - Scenario A) and distributed clusters based on Cloud servers (Scenario B). Here we assume that S represents a significant portion of the total running time, such that both scenarios would be a viable and effective way to handle the required computations.

With the assumption that the need for memory access and complexity in the analysis model remain the same in both scenarios, the computation running time depends on the clock frequency of the processors, memory access time, network's capacity to transmit data in unit time (bandwidth), and time needed to reach the data final destination (latency).

Scenario A would exploit symmetric multiprocessors with shared memory, which would require expensive SMP computers. Considering Moore's law on doubling number of transistors every other year (Rauber & Rünger, 2013, p.10) and our potentially increasing workload, this non-trivial investment might not be sustainable in the long term and inadequate. In the short and medium term however, Scenario A can achieve higher performance than Scenario B. In fact, single shared memory would make communication simpler without a need for data replication (Rauber & Rünger, 2013, p.19), thus lowering the overall latency and increasing the processing throughput.

On the other hand, Scenario B is easy and quick to procure and might not need a large capital investment at first hand. In the long term, if the needs of servers increase further, the company would then simply upgrade its subscription, since it would not be in change of upgrading the cloud hardware itself. Yet, considering the large data flow (1300 GB per day) and longer time needed for processing non-local data (Rauber & Rünger, 2013, p.17), Scenario A seems more feasible than Scenario B when it comes to minimize data movement and size of data blocks exchanged. Moreover, Scenario B might be prone to internet connectivity problems, and so Scenario A would be a secure alternative against possible slowdowns and/or interruptions of the company's internet infrastructure.

2 Problem 2

The code listing for problem 2 is added as a separate file. The graph for the speedup is included in Figure 1 graphs the speedup, calculated in terms of samples per second, of the `mp-pi-montecarlo-pool.py` running with different parallel workers. The results were obtained by averaging 30 repetitions of the program with an accuracy requirement of 0.0001. During each simulation, the size of the samples was 50.

As we can see, the speedup improves with an increasing number of parallel workers up until 16 ($p = 16$). Since the speedup is measured by the amount of samples which were computed per second, as opposed to exact speedup of the simulation, having more than 16 workers seems to not reduce the total running time enough in relation to the amount of generated samples. We believe we might be able to get a better speedup for 32 cores if we would further restrict the accuracy requirement, such that it would require a higher amount of samples.

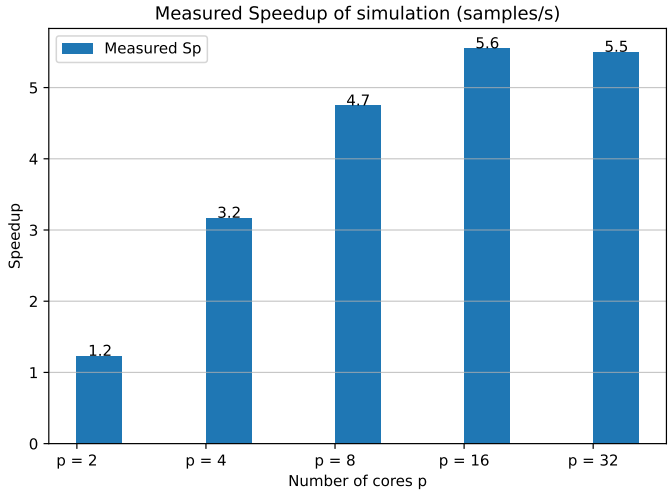

Figure 1: Measured speedups, in samples/s, of the simulations.

3 Problem 3

3.1 K-means Building blocks

Listing 1 show the Python function implementing the K-means algorithm. To better show our annotations, we removed all the code related to logging and the centroid variation calculations, as it is not used as a stopping criterion, in our case. The first part initializes a vector storing the cluster each point is assigned to and set as initial centroids k random data.

For the rest of the code, we highlight three code sections and also explain the reasons for eventually applying parallelization. Code Section 1 comprises the main refinement loop: the candidate centroid points are updated `nr_iter` times. The second Code Section 2 is inside the aforementioned loop and for each point, gets the closest cluster and keeps track of how many points are assigned to that cluster. Lastly, in Code Section 3 each new centroid point is computed as the average value of all the data points assigned to the corresponding cluster index.

Regarding applying parallelization techniques instead, since at each refinement iteration of the K-means algorithm (Code Section 1), we calculate the distance of each data point to its closest centroid (Code Section 2), each iteration *depends* on the previous one. Because of that, we cannot apply parallelization to Code Section 1 and therefore must be executed sequentially.

On the other hand, both Code Sections 2 and 3 *can* be parallelized across a suitable number of workers $w < N$. In fact, in Code Section 2 the vector `centroids` is only read and never modified, so it can be safely shared. Moreover, each worker needs to modify only its own subset of elements of the vector of cluster indexes. Finally, each worker can generate and then return a local vector of cluster counters to be then accumulated in the main thread.

A similar reasoning can be applied to Code Section 3: each worker can generate and return its own copy of the new centroid vector. Once the workers complete, the main thread would then aggregate the centroids copies and finally divide them by the cluster sizes.

```
def kmeansSerial(k, data, nr_iter=100, p=1):
    N = len(data)
    # The cluster index: c[i] = j indicates that i-th datum is in j-th cluster
    c = np.zeros(N, dtype=int)
    # Choose k random data points as centroids
    rand_idx = np.random.choice(np.array(range(N)), size=k, replace=False)
    centroids = data[rand_idx]
    # ==========================================================================
    # Code Section 1:
    # * The centroids vector is iteratively updated, meaning that the value of
    #   centroids depends on the previous iteration. Moreover, it is used to
    #   calculate the centroids distances, so the iterations must be executed
    #   sequentially.
    # * c (i.e. the cluster indeces) is shared and updated at each iteration.
    # ==========================================================================
    for j in range(nr_iter):
```

```
        # ============================================================
        # Code Section 2:
        # * For each point, get the closest cluster and keep track of how many
        #   points are assigned to that cluster.
        # * The centroids vector is shared, but never modified, so a copy can
        #   be passed to the workers without creating conflicts.
        # * c (i.e. the cluster indeces) is also shared, but each worker only
        #   modifies a certain pool of indeces.
        # * cluster_sizes represents a counter, therefore it can be updated
        #   locally by each worker and then accumulated afterward (in the main
        #   thread).
        # ============================================================
        # Assign data points to nearest centroid
        cluster_sizes = np.zeros(k, dtype=int)
        for i in range(N):
            cluster, dist = nearestCentroid(data[i], centroids)
            c[i] = cluster
            cluster_sizes[cluster] += 1
        # ============================================================
        # Code Section 3:
        # * Each new centroid is the average value of all the data points
        #   assigned to the corresponding cluster index.
        # * Like cluster_sizes, also the centroids vector can be updated
        #   locally by each worker and then accumulated and averaged
        #   afterward (in the main thread)
        # ============================================================
        # Recompute centroids
        centroids = np.zeros((k, 2)) # This fixes the dimension to 2
        for i in range(N):
            centroids[c[i]] += data[i]
        centroids = centroids / cluster_sizes.reshape(-1, 1)
    return c
```

Listing 1: Annotated Python code for the K-means algorithm.

3.2 Running Time Measurements

We measured the running time of the K-means algorithm clustering 10,000 data points in 4 clusters. The total running time was roughly of 15.3 seconds. Code Section 2, which includes the `nearestCentroid()` function, please see previous subsection, took a total of 14.01 seconds, 92% of the total time ($f_{sec2} = 0.92$). Code Section 3, *i.e.* the centroids recomputation, please see previous subsection, instead took in total 1.29 seconds, 8% of the total time ($f_{sec3} = 0.08$). This shows that the purely sequential part, *i.e.* Code Section 1, has a negligible running time.

3.3 Amdhal's Law

Given the estimated fractions of time f_{sec2} and f_{sec3} of the two candidate parallel sections, reported in the previous subsection, we can calculate the speedups via Amdhal's law as

follows:

$$Sp(w) = \frac{1}{1 - f_{sec} + \frac{f_{sec}}{w}}$$

where w represents the number of parallel workers.

A summary of the calculated speedups in reported in Table 1. For Code Section 2, we have $Sp(w = 4) = 3.19$ and $Sp(w = 8) = 5.006$. For Code Section 3 instead, we have $Sp(w = 4) = 1.068$ and $Sp(w = 8) = 0.081$. As Amdhal's law confirms, the highest speedup comes from parallelizing Code Section 2, since $f_{sec2} > f_{sec3}$.

Table 1: Speedups of the candidate parallel Code Sections.

Code Section	f	$Sp(4)$	$Sp(8)$
Code Section 2	0.92	3.19	5.006
Code Section 3	0.08	1.068	1.0081

References

[1] Rauber, Thomas, and Gudula Rünger. *Parallel programming*, Berlin, Germany: Springer, 2013.

[2] Schliep, Alexander. *DIT065/DAT470 Computational techniques for large-scale data: Lecture 2 Notes*, 2022. Available at https://chalmers.instructure.com/courses/18200.