Parallel k-means Clustering

SF2568 – Parallel Computations for Large-Scale Problems

Lukas Bjarre **Gabriel Carrizo**
lbjarre@kth.se **carrizo@kth.se**

Abstract

This report details the implementation and analysis of a parallelisation of the k-means clustering method. A serial version of Lloyd's algorithm for computing the k-means clustering was analysed and a parallelisation stategy of dividing data points over multiple processes was proposed. The parallel implementation was a single program multiple data scheme on distributed memory using the Message Passing Interface (MPI) standard. The implementation was tested on the super computer cluster Tegner located at the PDC Center for High Performance Computing at the Royal Institute of Technology in Stockholm. The implementation showed a great speedup for the computation taking the computational time from 300 s to 18 s with 20 processors on one large image.

1 Introduction

1.1 Clustering

Clustering is described in [1] as the grouping of similar objects. Hartigan explains that two objects are similar if their separate observations could be of the same object. Using the eample of planets, Hartingan explains that two observed objects are planets if their observations are similar enough in significant features. Consider for example the features: shape and light intensity, versus color to discern planets from stars.

1.2 Parallel Computations

Today, clustering is an important tool for automatic grouping or preprocessing of data in data science. With the emergence of *big data* where the number of data points and dimensions of the data are very large these algorithms can be computationally costly. Traditionally these algorithms are written in a linear manner, where *instruction A* is followed by *instruction B*, even if they operate on disjoint data. Modern processors with multiple cores have the capability of executing multiple instructions, or processes, at the same time. This enables some problems to be computed more efficiently, however one prerequisite is the possibility of dividing the problem into disjoint subsets that can be processed individually. This class of problem falls into the parallel category whilst the opposite is classed as a sequential problem. A sequential problem is one in which each section of the code is dependent on the previous sections output in order to proceed with the computations. Many problems are partly parallel and sequential and can be subdivided into both categories. Parallel computations are more complex than sequential and are usually preserved for problems where the computation time is important or where the computation time can be reduced substantially.

Parallel algorithms can be divided into two sub classes: shared vs. distributed memory. Distributed memory implementations require communication between processes where data and results of computations are exchanged or gathered. Although these transactions take relatively long time to perform, distributed implementations are prefered as shared memory implementations suffer from problematic *race conditions*.

1.3 Aim

This project proposes a distributed parallel implementation of a clustering algorithm to reduce the computation time for more efficient processing of data. The project aims to reduce the computation time of a modern interpretation of a fairly simple clustering algorithm *K-means clustering* as a potential stepping stone to more complex algorithms.

2 Theory

2.1 k-means clustering

k-means clustering is a data clustering method which clusters input data from the data set $\mathcal{X}$ into k different classes. The classes are represented by the class means $\boldsymbol{\mu}_i$ and points are considered to be in a class S_i if the squared distance to the class mean is the minimum compared to the squared distance to the other class means. Formally:

$$S_i = \{\boldsymbol{x} \in \mathcal{X} : \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2 \leq \|\boldsymbol{x} - \boldsymbol{\mu}_j\|^2, \forall 1 \leq j \leq k\}$$

A clustering method aims to find a selection of these classes $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ which divides the data points in some favorable way. k-means finds the placement of the class means by minimization of the summed squared distance of all class points to the class mean for all k classes:

$$\mathcal{S}_{k\text{-means}} = \arg\min_{\mathcal{S}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2$$

A common algorithm to find this is Lloyd's algorithm [2], which iteratively classifies points according to current class means and updates them with the average of all classified points until convergence. Algorithm 1 describes this procedure in pseudocode. This algorithm requires multiple loops for each iteration. The most costly part is the classification of all data points, which requires computation on all d dimensions of the data set for all k classes over all n points, giving each iteration a complexity of $O(nkd)$.

Algorithm 1: Lloyd's algorithm for finding the k-means clustering class means.

Input: Data points $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ with $\boldsymbol{x}_i \in \mathbb{R}^d \; \forall \boldsymbol{x}_i \in \mathcal{X}$, number of clusters k
Output: Class means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k$

initialize $\bar{\boldsymbol{\mu}}_i$ on random points in $\mathcal{X}$ $\forall i = 1, \ldots, k$
while $\boldsymbol{\mu}_i \neq \bar{\boldsymbol{\mu}}_i \; \forall i = 1, \ldots, k$ **do**
  for $i = 1, \ldots, k$ **do**
    $\boldsymbol{\mu}_i \leftarrow \bar{\boldsymbol{\mu}}_i$
    $\bar{\boldsymbol{\mu}}_i \leftarrow 0$
  forall $\boldsymbol{x} \in \mathcal{X}$ **do**
    class $\leftarrow \arg\min_k \|\boldsymbol{x} - \boldsymbol{\mu}_k\|^2$
    $\text{count}_{\text{class}} \leftarrow \text{count}_{\text{class}} + 1$
    $\bar{\boldsymbol{\mu}}_{\text{class}} \leftarrow \bar{\boldsymbol{\mu}}_{\text{class}} + \boldsymbol{x}$
  for $i = 1, \ldots, k$ **do**
    $\bar{\boldsymbol{\mu}}_i \leftarrow \frac{\bar{\boldsymbol{\mu}}_i}{\text{count}_i}$
    $\text{count}_i \leftarrow 0$
return $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k\}$

2.2 Parallelisation strategy

Algorithm 1 shows a lot of possibility of parallelisation due to the large number of independent calculations. For every iteration all the points needs to be reclassified to the new means, for which every point is independent of the other points. Computing the new class means consist of adding up the classified points over all d dimensions as well as the total count, which are additions that can be done firstly on separate processors and finally reduced over all processors.

Overall we get a execution time for each iteration T_{iter} that with the number of processes P that roughly scales as

$$T_{\text{iter}} = t_{\text{calc}} \frac{nkd}{P} + t_{\text{comm}} k(d+1) \log P$$

Noteworthy for the analysis of this performance is that n will be much larger than k and d in most applications. k and d normally lie in the 10^1-10^2 range, while n can go up to 10^6-10^9

easily. The reduction of the calculation time will therefore affect the overall computation time more than the increasing communication time.

Algorithm 2 shows the minor change to make algorithm 1 into a parallel algorithm. The main idea is to distribute the data set over all the processors and allow them to classify their local data points in parallel.

Algorithm 2: Parallel version of Lloyd's algorihtm

Input: Data points $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ with $\boldsymbol{x}_i \in \mathbb{R}^d \; \forall \boldsymbol{x}_i \in \mathcal{X}$, number of clusters k, number of processes P

Output: Class means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k$

Distribute dataset into local datasets $\mathcal{X}_i \; \forall i = 1, \ldots, P$
initialize $\boldsymbol{\mu}_i$ on random points in $\mathcal{X} \; \forall i = 1, \ldots, k$
while $\boldsymbol{\mu}_i \neq \bar{\boldsymbol{\mu}}_i \; \forall i = 1, \ldots, k$ **do**
    for $i = 1, \ldots, k$ **do**
        $\boldsymbol{\mu}_i \leftarrow \bar{\boldsymbol{\mu}}_i$
        $\bar{\boldsymbol{\mu}}_i \leftarrow 0$
    forall $\boldsymbol{x} \in \mathcal{X}_i$ **do**
        class $\leftarrow \arg\min_k \|\boldsymbol{x} - \boldsymbol{\mu}_k\|^2$
        $\text{count}_{\text{class}} \leftarrow \text{count}_{\text{class}} + 1$
        $\bar{\boldsymbol{\mu}}_{\text{class}} \leftarrow \bar{\boldsymbol{\mu}}_{\text{class}} + \boldsymbol{x}$
    for $i = 1, \ldots, k$ **do**
        reduce_sum($\bar{\boldsymbol{\mu}}_i$)
        reduce_sum(count_i)
        $\bar{\boldsymbol{\mu}}_i \leftarrow \frac{\bar{\boldsymbol{\mu}}_i}{\text{count}_i}$
        $\text{count}_i \leftarrow 0$
return $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k\}$

3 Method

3.1 Input data

k-means clustering is not strictly limited to any type of input data. However, a use case large enough to motivate the use of a parallel environment was wanted. A simple way of making the input data large is to use large pictures, where the clustering method is working to cluster pixels into representative colors. Altough containing just 3 dimensions for the color channels in the pictures, pixel density can easily reach extremely large ranges. This project settled on a 72-megapixel image which can be seen in fig. 1.

3.2 Reading data

Runs of early versions of the code indicated that reading the file sequentially and scattering the data to the processes took too long. A solution for reading the data in parallel was developed so the sequential read time would not become a bottleneck for the code. For this, MPI IO's function MPI_File_read_at_all()[1] was utilized. However, since the function reads bytes in a document rather than rows, the data was pre-formatted such that all indides of a column were the same length. Each process then reads the first row to count how many characters are in a line. This allows calculating how many characters the different processes should read, n_p, by performing the following calculations:

$$n_p = \frac{N + P - p - 1}{P} l \tag{1}$$

[1] Documentation-https://www.open-mpi.org/doc/v2.0/man3/MPI_File_read_at_all.3.php

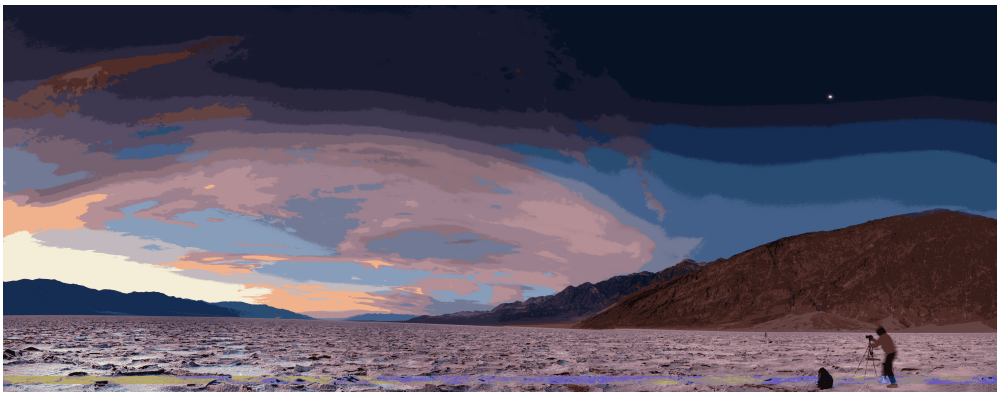
where N is the total number of lines in the file, P is the number of processes, p is the rank of the process and l is the length of a row. Once read, each process converts the contents of its partition into floats and stores the result in an array.

4 Results

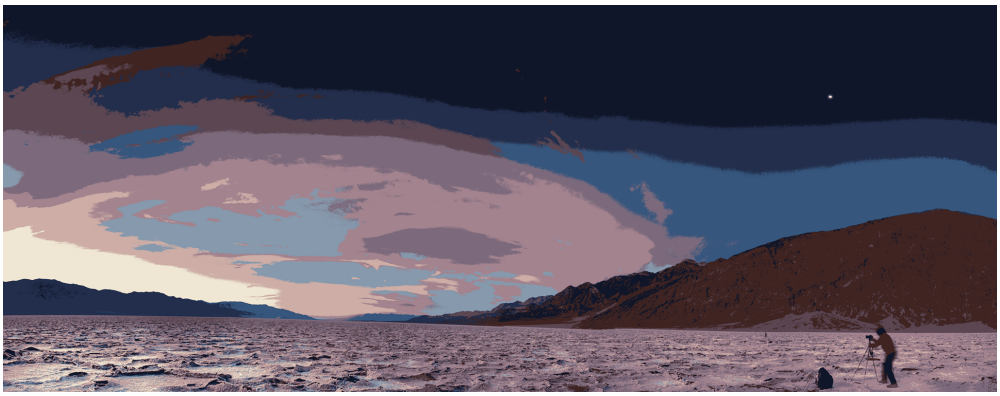
4.1 Image compression

In fig. 1 both the original image used as the input data and the resulting image after the k-means clusters had been found for $k = 24$ and $k = 10$.

(a) Before k-means compression



(b) After k-means compression with $k = 24$ clusters



(c) After k-means clustering with $k = 10$ clusters

Figure 1: The input image and its outputs after clustering

4.2 Computation time

Table 1 and section 4.2 shows the elapsed time during the computations on the picter done at Tegner for increasing ammount of processes. The image seen in fig. 1 was used as the input for all of the plotted times with k set to 10. For $p = 1$ the serial version of Lloyd's algorithm, algorithm 1, was used, while for all $p > 1$ the parallel version, algorithm 2, were used. 4-5 runs where used for each process setup and used to create a statistical mean and standard deviation, which are plotted as the area in the figure. We can see that the computation time decreases in a exponential fashion as expected.

Table 1: Computational time for the parallel implementation

Processes	T_{comp}	σ
1	312	0.63
2	184.25	3.49
4	90.4	0.49
6	80.75	2.17
8	40.2	15.38
10	44.2	7.14
12	47.6	2.24
14	32	2.83
16	32.6	3.88
18	32	0
20	18.6	3.38
22	18.2	18.16

5 Discussion

We have presented a parallel implementation of the famous k-means algorithm. The preformance of the algorithm is accelerated when used with more processes but the implementation does not seem to match Amdahl's law (as can be seen in section 4.2). However, since the kernels are randomly initiated and the algorithm runs until convergence the measurements are not completely reliable. We decided to run the algorithm multiple times and calculte a mean and a variance but we only had time to run the algorithm approximately 5 times for each setting which proved insufficient. The effects of this can be seen in fig. 2c where for example $p = 10$ has high standard deviation from an outlier. This in combination with time's standard settings only showing seconds means that the results re affected by rounding errors. This can be seen especially in fig. 2c where $p = 18$ has zero standard deviation.

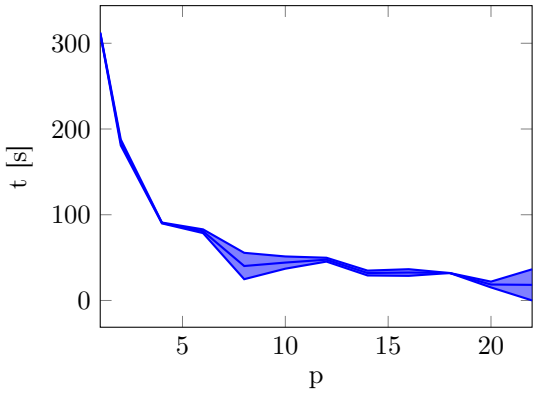
Another noteworthy feature of k-means that we discovered when we ran the algorithm on an image was that it was compressed. The resulting image file was compressed to approximately $\frac{1}{4}$ of its original size.

As can be seen in section 4.2 there is a hiccup for $p = 10$ which we have no explanation for.

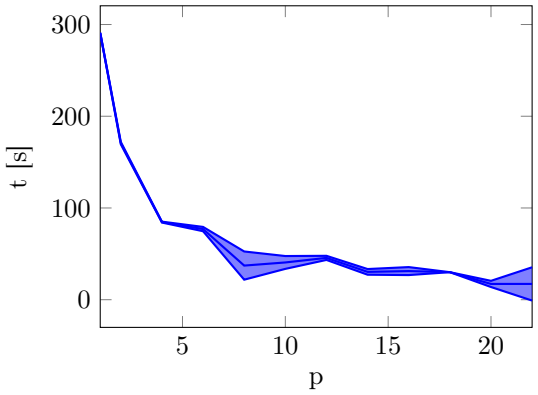
References

[1] John A Hartigan. Clustering algorithms. pages 1–14, 1975.

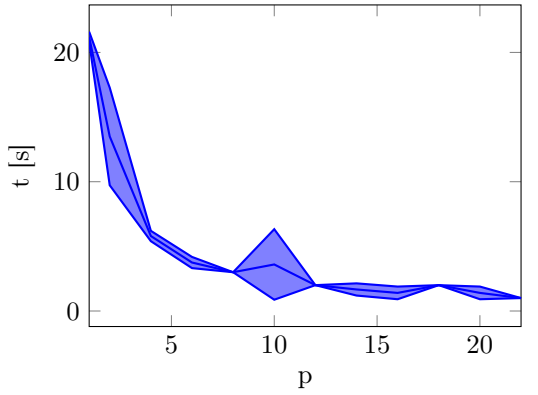
[2] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.



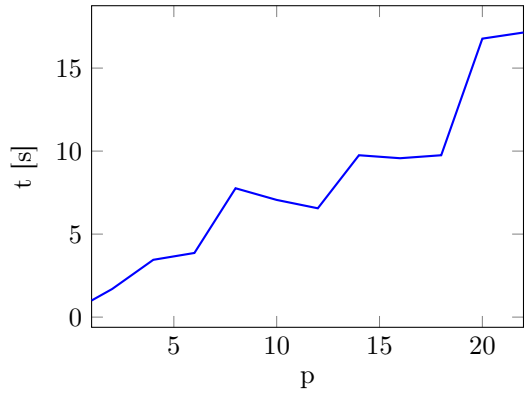
(a) Total execution time



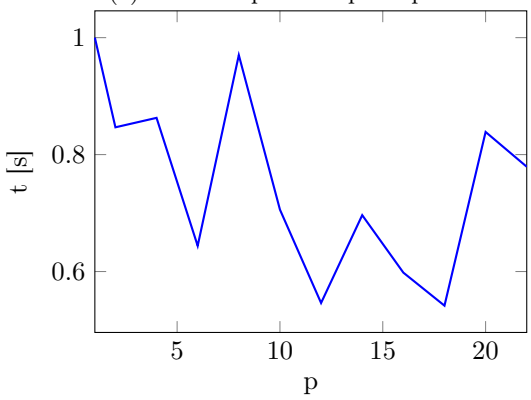
(b) Execution time for k-means parallel



(c) Execution time for reading the data in parallel



(a) Absolute parallel speedup



(b) Relative parallel speedup