Sequential and Parallel Implementation of K-Means Clustering algorithm in Python/Go and visualization in Pharo

k-Means clustering is an **unsupervised** machine learning algorithm that finds a fixed number (k) of clusters in a set of data. A **cluster** is a group of data points that are organized together due to similarities in their input features. When using a K-Means algorithm, a cluster is defined by a **centroid**, which is a point at the center of a cluster. Every point in a data set is part of the cluster whose centroid is most closely located. So, k-Means finds k number of centroids, and then assigns all data points to the closest cluster, with the aim of keeping the centroids small (we tend to minimize the distance between points in one cluster, so that they make compact ensemble and to maximize distance between different clusters).

Given a set of observations (x1, x2, ..., xn), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k (≤ n) sets S = {S1, S2, ..., Sk} so as to minimize the within-cluster sum of squares. Formally, the objective is to find:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \left\| \mathbf{x} - \boldsymbol{\mu}_i \right\|^2 = \underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

where µi is the mean of points in Si. This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \left\| \mathbf{x} - \mathbf{y} \right\|^2$$

Sequential approach

1. Cluster the data into k groups where k is predefined
2. Select k points at random as cluster centers
3. Assign objects to their closest cluster center according to some distance function (for example *Euclidean distance*)
4. Calculate the centroid or mean of all objects in each cluster
5. Repeat steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds

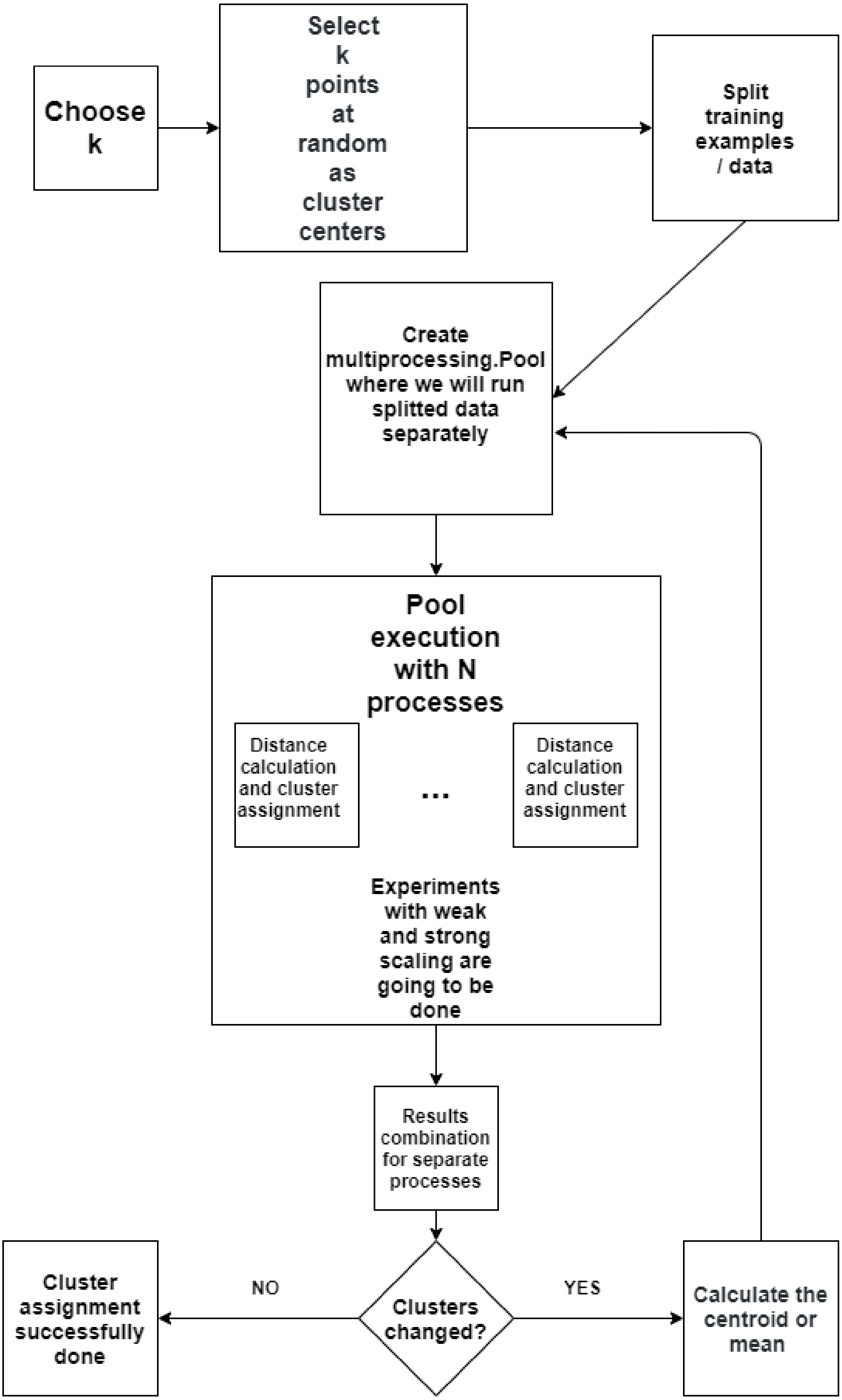
Parallel approach

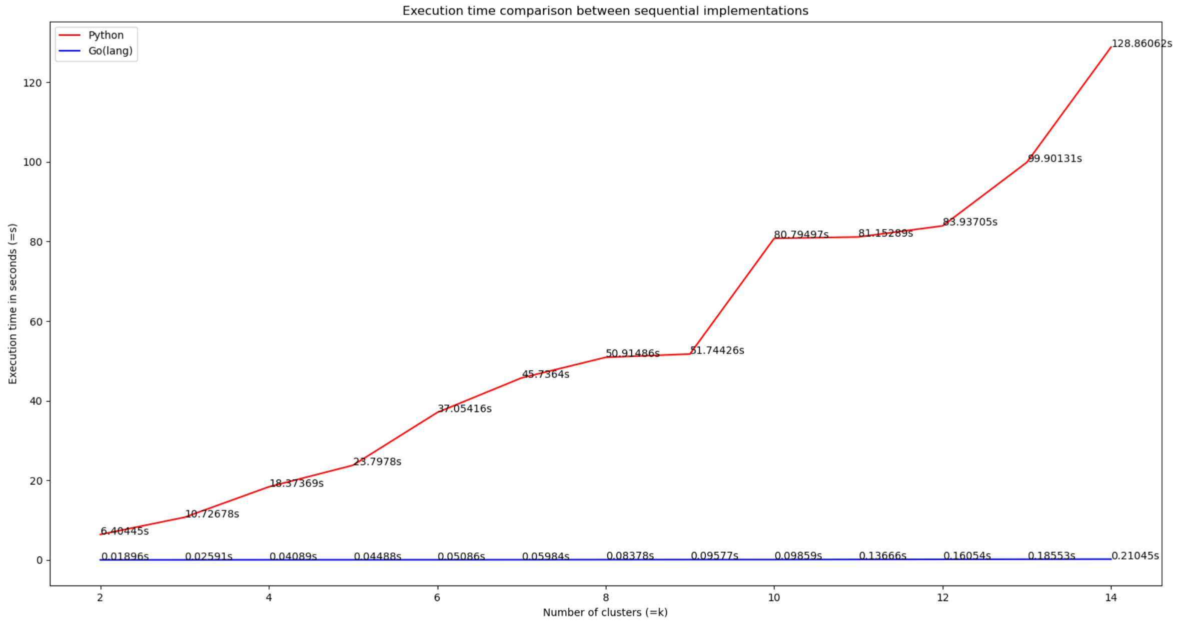
Main **motivation** for parallel approach is the fact that k-means clustering performance decreases when we increase number of training examples and when we decide to have a larger k.

The main objective of this project is to improve performance of k-Means clustering algorithm by splitting training examples into multiple partitions and then we calculate distances and assign clusters in parallel. After that, cluster assignments from each partition are combined to check if clusters changed. For iteration I, if clusters changed in iteration (I -

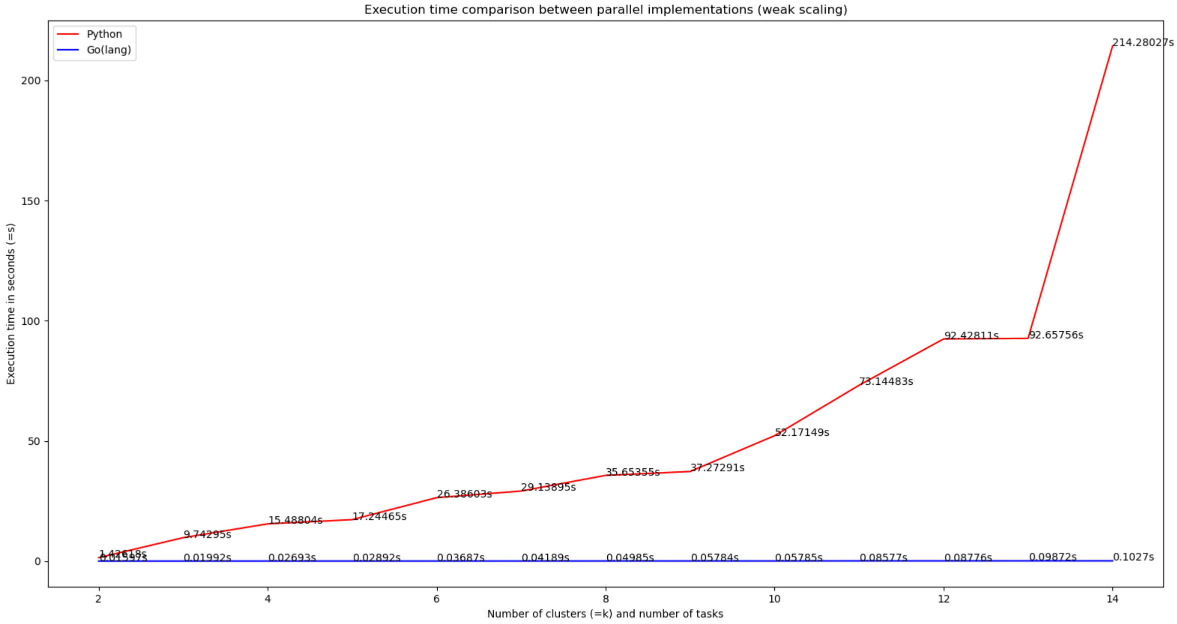
1), we need to recalculate centroids, else we are done.

The whole process is shown on the following diagram (generated with: Flowchart Maker):

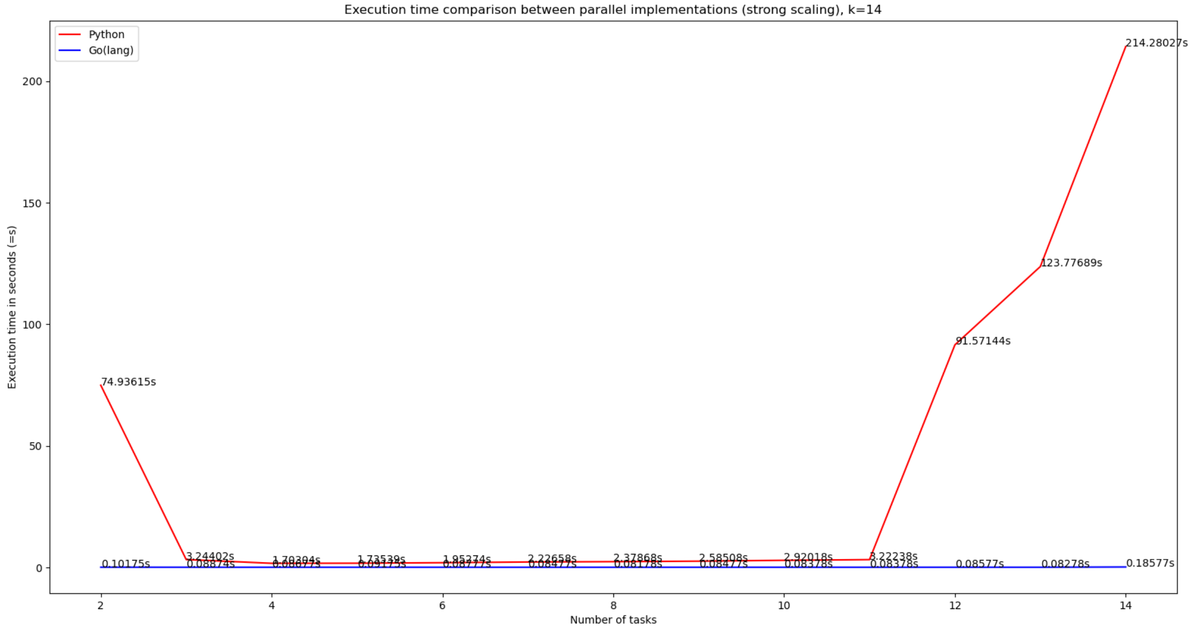

Choose k
Select k points at random as cluster centers
Split training examples / data
Create multiprocessing.Pool where we will run splitted data separately
Pool execution with N processes
Distance calculation and cluster assignment
...
Distance calculation and cluster assignment
Experiments with weak and strong scaling are going to be done
Results combination for separate processes
Clusters changed?
NO
YES
Cluster assignment successfully done
Calculate the centroid or mean



On the plot above, we can see execution time comparison in seconds between sequential implementations (red line = Python and blue line = Go). **Go is typically 40 or more times faster than Python** because Go compiles **directly to native machine code**, while Python's runtime dynamism is difficult to optimize for speed.



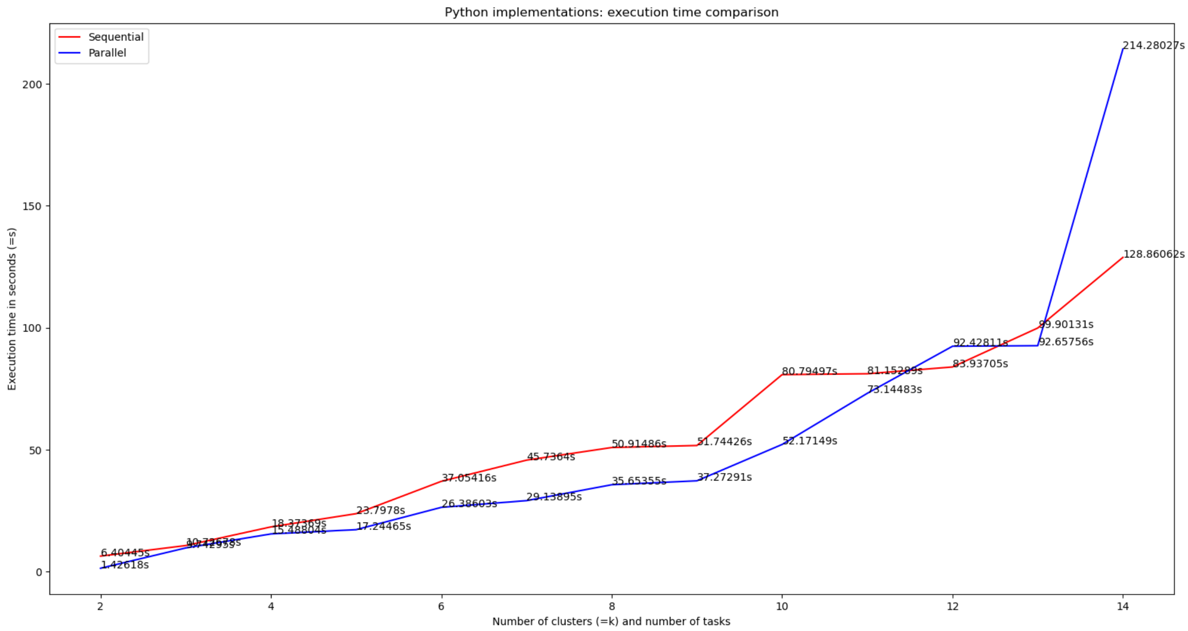
On the following plot, we can see execution time comparison between parallel implementations (**weak scaling**) between Python (red line) and Go (blue line). As we increase the number of clusters / number of tasks (=k) on X-axis, execution time on Y-axis rises which is logical, if we have more clusters we need more time to operate, but with increasing number of tasks in parallel we can work with C clusters by not waiting too much more than by working with C – 1 clusters. In Go everything is pretty fast, and in Python when k = 14 we get to the point where we have too much tasks for this problem and time drastically increases (**peak for the problem**)



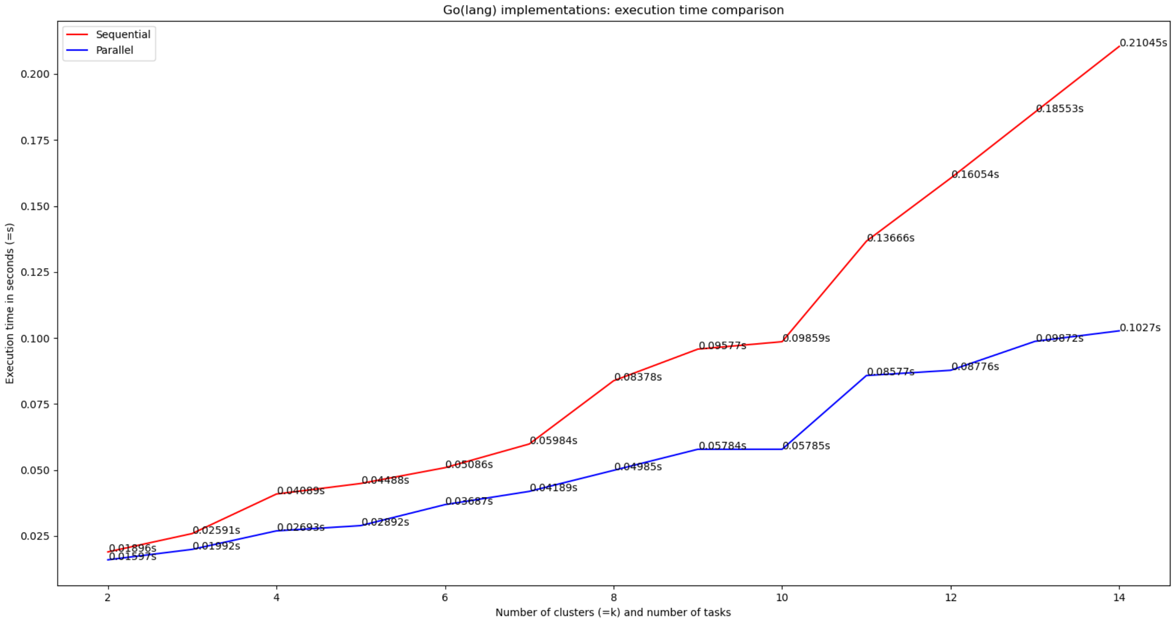
On this plot, we can see execution time comparison between paralell implementations (**strong scaling**) with a fixed k (=14) number of clusters, where we analyzed how increase in number of tasks (on X-axis) had impact on execution time in seconds (on Y-axis). The main conclusion is that Python has processes which **don't share same memory** and he loses a decent amount of time for context-switching, while Go(lang) has **Goroutines which share it** and he loses less time because of that (time increases, but not as in Python).

One challenge when addressing this problem was that initializations of centroids have an effect on result output, so when comparing Python and Go, and their implementations, all of these plots represent results with same init-centroids (these initial values can be seen in https://github.com/NikolaZubic/K-MeansClusteringPythonGoPharo/blob/master/Python/plots/sequential/pythonGoComparison.py). Conclusion is when we increase number of clusters (=K) and dataset size that **parallel implementation shows its power in terms of speed when compared to sequential implementation in same language**, but **Golang is always faster than Python**.

As we can see, one of the differences between Python and Go is that Go works more on low-level and Python has more useful (but not necessary efficient) libraries for AI / ML field than Golang. We know that Python is **interpreted** and dynamically typed language, while Go is **compiled** and **statically typed** language.



This plot represents execution time comparison of Python implementations where red line is sequential and blue line is parallel implementation. We can see that after certain number of tasks (here, on College.csv dataset, >12) there is no purpose of increasing number of tasks, since Python loses a lot of time on context-switching.



On the last plot, we can see execution time comparison of Go(lang) implementations where red line is sequential and blue line is parallel implementation. We can see that sequential implementation takes more time even for K = 14 for this dataset, because **Goroutines share same memory** and don't lose too much time on context-switching, so it is tolerable to use for K values that are even bigger than 14.

System configuration (HP Omen 17-an0xx):

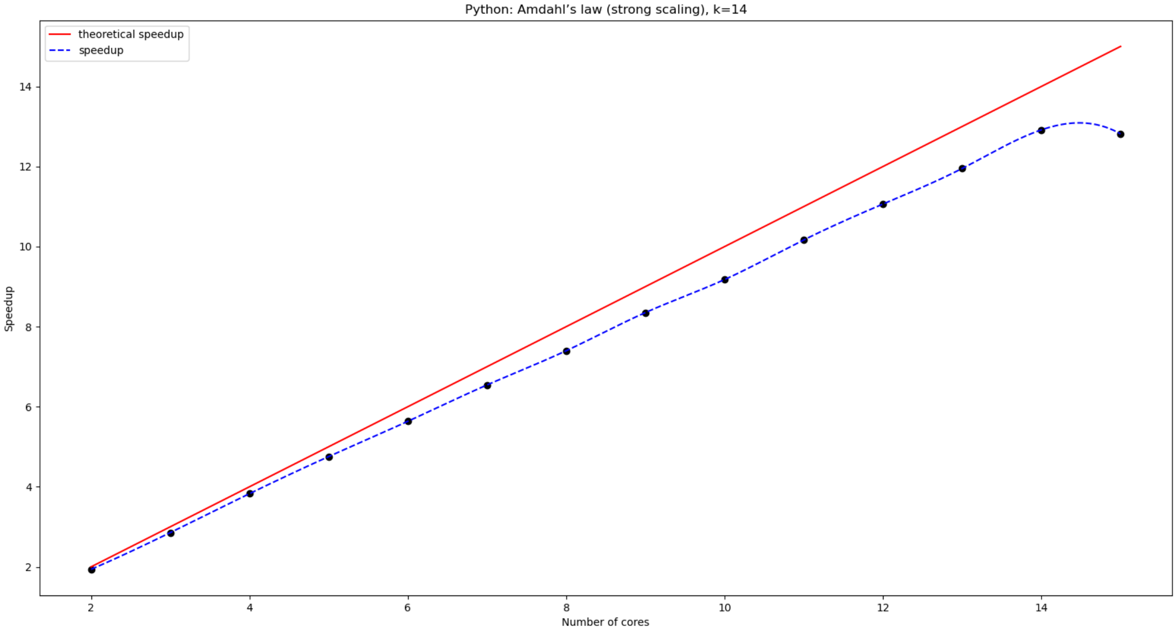
- Intel Core i7 7700HQ 2.8GHz, 8 CPUs/Logical Processors and 4 Core(s)
- 16GB DDR3 Memory
- 256GB Solid State Drive
- NVIDIA GeForce GTX 1070 8GB VRAM, TOTAL MEMORY: 16 GB

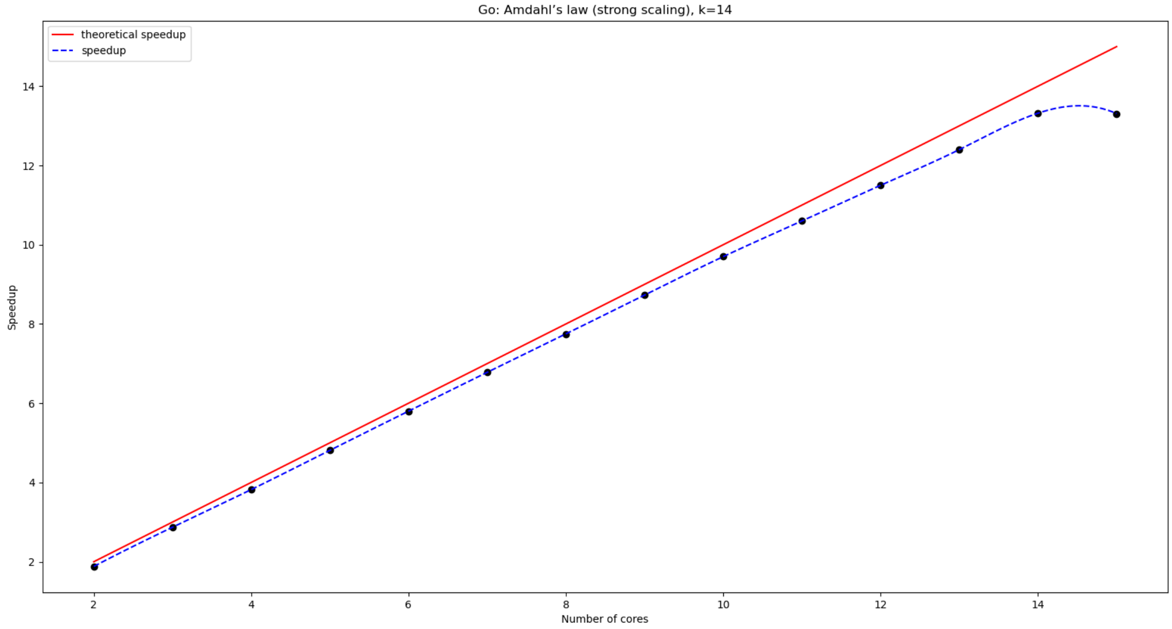
When addressing scalability, S will represent the proportion of execution time spent on the sequential part and P is the proportion of execution time spent on the part that can be parallelized. In this project: **S ~ 2.5% and P ~ 97.5 %**

Assuming all of this, we conclude that in **Amdahl's law** (strong scaling) theoretical speedup is $1 / (S + P/N) = 1 / (0 + 1/N) = 1 / (1/N) = N$ (equal to number of cores). In **Gustafson's law** (weak scaling) theoretical speedup is $S + P * N = 0 + 1 * N = N$.

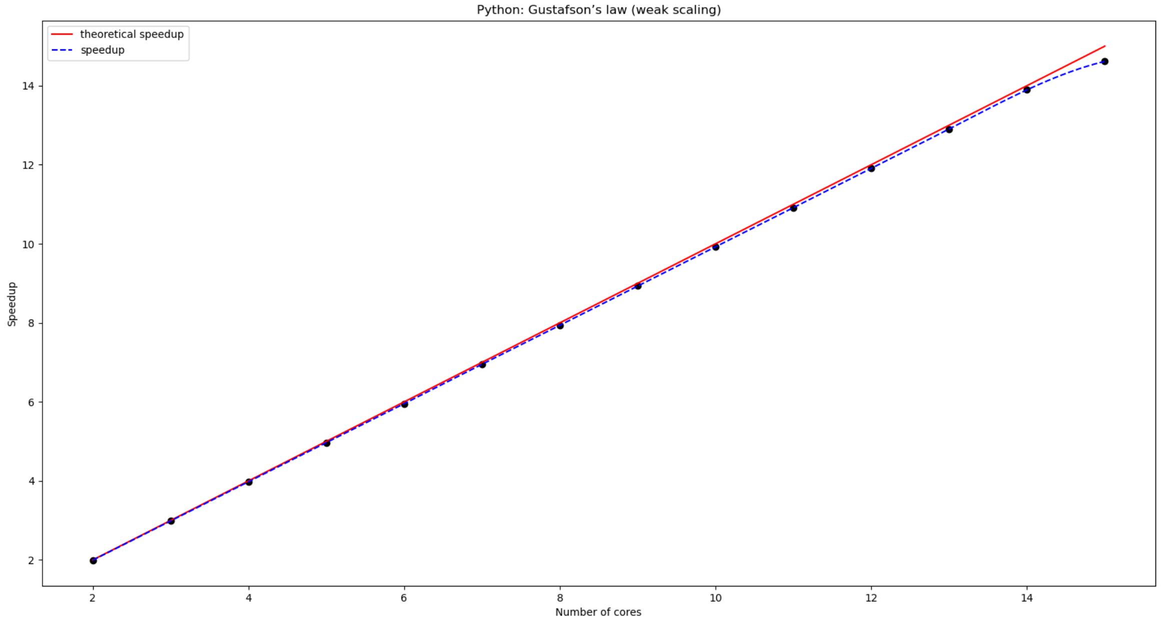
So, in both cases theoretical speedup **equals N**.

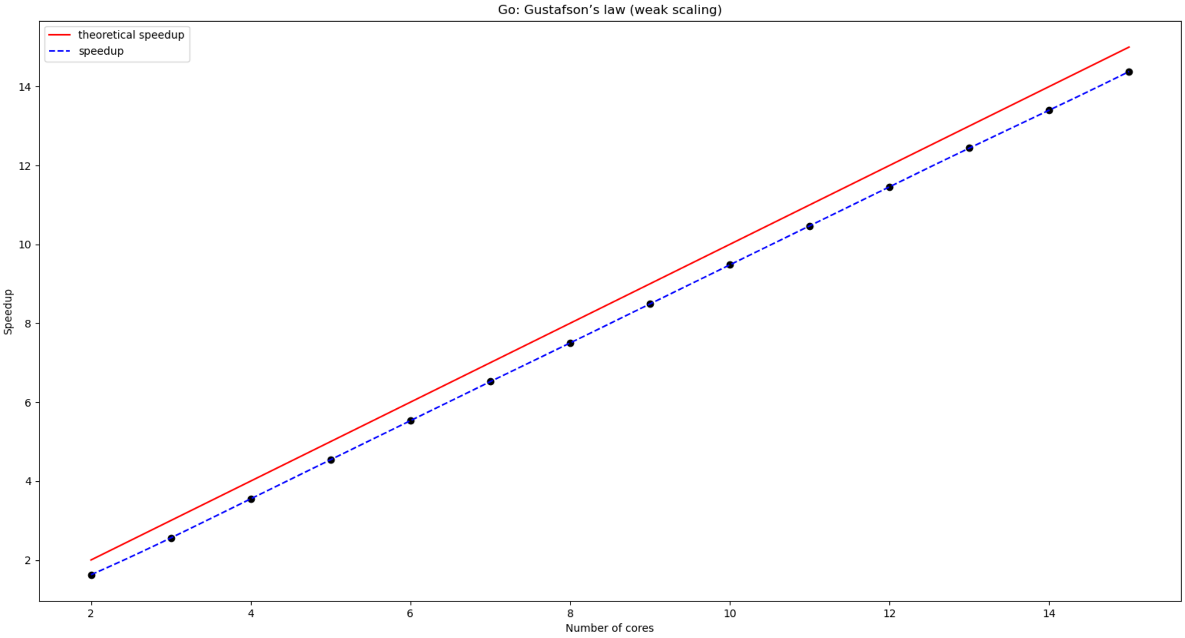
On following **two** plots we can see strong scaling experiments obtained in Python and Go. We can conclude that after 14 number of cores there is no significant speedup, so it doesn't make sense to increase it after 14, for fixed load where K = 14. Also, Go scales slightly better.





On the last two plots that analyze weak scaling, on X-axis we set **number of cores to be equal to number of clusters (=K)** and we can see that by increasing number of cores along with the number of clusters we get speedup which is almost linear and decreases really slowly.




Go: Gustafson’s law (weak scaling)
theoretical speedup
speedup
Speedup
14
12
10
8
6
4
2
Number of cores
2
4
6
8
10
12
14