Distributed Data Analytics Lab

Exercise 4

Sebastián Pineda Arango Mtr. Nr. 246098

In this laboratory, we want to perform stochastic gradient descent in a distributed way using to datasets:

- **Dataset 1:** Dynamic Features of VirusShare Executables Data Set [1]
- **Dataset 2:** KDD Cup 1998 Dataset [2]

First, we want to discuss the stochastic gradient descent performed in a single processes. Moreover, we want to compare the performance of the implemented distributed MPI algorithms. At the end, we will be able to show that by dividing the data among different processes, it is possible to achieve a gain in execution time.

1) Sequential stochastic gradient descent

The sequential gradient descent, perform iterations by updating the gradient of the parameters only with a sample at a time. After a whole ran on the training set, the algorithm finishes.

If $\boldsymbol{x}_i$ is a sample, with label y_i, then, the gradient of the loss function given this sample for a linear regression model with parameters $\boldsymbol{\beta}$ is given by:

$$\frac{\partial L}{\partial \boldsymbol{\beta}} = -\boldsymbol{x}_i^T(y - \boldsymbol{x}_i\boldsymbol{\beta}) \qquad Equation\ 1$$

Therefore, a stochastic update consist in:

- Picking a random sample x_j
- Computing the gradient using the last equation and the sample
- Updating the parameters

$$\boldsymbol{\beta} = \boldsymbol{\beta} - \mu\frac{\partial L}{\partial \boldsymbol{\beta}} \qquad Equation\ 2$$

Where μ is the learning rate.

- **Code for sequential stochastic gradient descent**

We show the code for stochastic gradient descent. Comparisons are carried out later on.

```
import numpy as np
import os
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
from mpi4py import MPI
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
```

```
import pandas as pd
import time
import sys


#initializing important variables
time_per_epoch = []
RMSE_train_history = []
RMSE_test_history = []
lr = 0.0000025
epochs = 20

name = sys.argv[1]

#reading datas
if(name[0]=="K"):
    X, y = load_data("KDD_CUP")
    print("Loading KDD CUP dataset...")
else:
    X, y = load_data("VIRUS")
    print("Loading virus dataset...")

X = np.hstack((np.ones((X.shape[0],1)),X))

#splitting data in train and test
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2,
random_state=42)

#initializing parameters
beta = np.matrix(np.random.rand(X.shape[1],1))

#scaling data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#formatting data
X_train = np.matrix(X_train)
y_train = np.matrix(y_train)
X_test = np.matrix(X_test)
y_test = np.matrix(y_test)

n_train_samples = X_train.shape[0]
n_test_samples = X_test.shape[0]
```

```
#performing stochastic grdient descent
for epoch in range(epochs):

    init = time.time()
    for i in range(X_train.shape[0]):

        idx = np.random.choice(X_train.shape[0],1)
        beta = beta - lr*grad(X_train[idx,], y_train[idx,], beta)

    #saving RMSE information
    elapsed = time.time()-init
    time_per_epoch.append(elapsed)
    RMSE_train_history.append(linear_loss (X_train, beta, y_train))
    RMSE_test_history.append(linear_loss(X_test, beta, y_test ))

#formating RMSE information
RMSE_train_history = np.array(RMSE_train_history)
RMSE_test_history = np.array(RMSE_test_history)
time_per_epoch = np.array(time_per_epoch)

#wirintg information to disk for further analysis
np.save(name+"_RMSE_train", RMSE_train_history)
np.save(name+"_RMSE_test", RMSE_test_history)
np.save(name+"_time", time_per_epoch)
```

2) **Distributed stochastic gradient descent**

For the distributed stochastic gradient descent setting, we perform following steps:

a) **Load and preprocessing of the data**
b) **Data distribution and parameters initialization**
c) **Parameters and stop flag sharing**
d) **Local stochastic gradient descent**
e) **Parameters aggregation and convergence checking**
f) **If stop condition is not met, go back to step c).**

In the following part, we explain step by step and show the python code to do.

Step-by-step

a) Load and preprocessing of the data

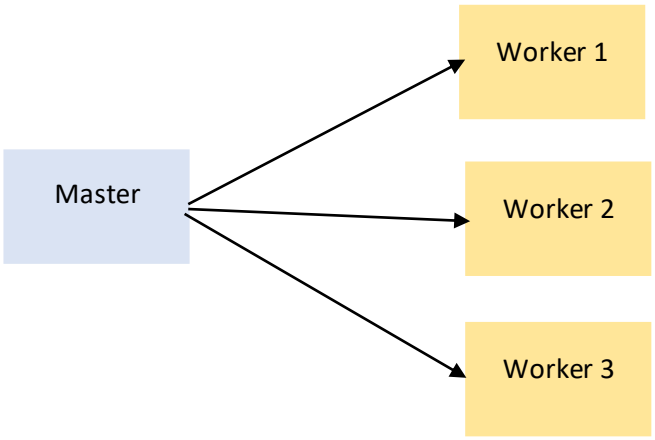
Since there are two datasets, we perform two different preprocessing steps on each one. The **master** is in charge of performing the load an preprocessing of the data.

- *Preprocessing step in Dataset 1*

  The dataset 1 (virus dataset), is in svmlight format. Therefore, we use scikit-learn function "load_svmlight_file" to read every file of the dataset. Later, the dataset is split in training and test subsets, and, finally, the features are standarized by using "StandardScaler" object from scikit-learn.

- *Preprocessing step in Dataset 2*

  The dataset 2 is in csv format, therefore, we use pandas to read it. Also, we select only the numerical columns as features. Moreover, we eliminate those columns which have more than 30% missing values and, afterwards, we drop all the rows that at least have one missing value in one of the features. Similarly to dataset1, we split the dataset in training and test subsets using "train_test_split" function from scikit-learn. For standardizing, we leverage scikit-learn library by using "StandardScaler".



The following implemented for splitting the data is called "load_data" and is defined as follows:

```
def load_data(dataset):

    """This functions loads the dataset to use in the processing depending
on the arguments"""

    if dataset=="KDD_CUP":

        #read data
```

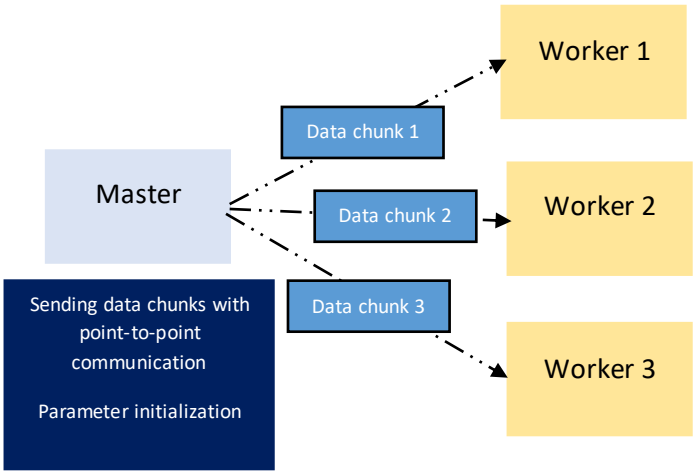
```
        data = pd.read_csv("cup98LRN.txt", sep=",")

        #selecting on the non-numerical columns
        numerics = ['int16', 'int32', 'int64', 'float16', 'float32',
'float64']
        newdf = data.select_dtypes(include=numerics)
        features = list(newdf.columns)

        #eliminating columns with many missing values
        col_na = newdf.isna().mean()
        col_remove = list(col_na[col_na>0.3].index)

        for feature in col_remove:
            features.remove(feature)

        #dropping rows with at least one missing value
        data_cleaned = data[features].dropna()

        #removing targets from the features columns
        features.remove("TARGET_D")
        features.remove("TARGET_B")

        #formatting output to array
        X = np.array(data_cleaned[features])
        y = np.array(data_cleaned["TARGET_D"]).reshape(-1,1)


    else:

        #referecing directory with the data
        path = os.path.join("dataset","")
        files = os.listdir(path)
        data = []

        #reading files which are in svmlight format
        X, y = load_svmlight_file(path+"/"+files[0])
        X= X.todense()
        for file_name in files:

            X_, y_ = load_svmlight_file(path+"/"+file_name, n_features=479)
            X_= X_.todense()
            X = np.vstack((X, X_))
            y = np.vstack((y.reshape(-1,1), y_.reshape(-1,1)))

    return X, y
```

On the other hand, the code used by the master to preprocess the data is the folowing:

```
if(name[0]=="K"):
        X, y = load_data("KDD_CUP")
        print("Loading KDD CUP dataset...")
    else:
        X, y = load_data("VIRUS")
        print("Loading virus dataset...")

    X = np.hstack((np.ones((X.shape[0],1)),X))

    #initializing flags and list to fill with perforamcne measurements
    stop_worker = [False]*num_workers
    MSE_train_list = [0]*num_workers
    MSE_test_list = [0]*num_workers
    MSE_train_diff = [1]*num_workers
    RMSE_train_history = []
    RMSE_test_history = []
    time_per_epoch = []

    #normalizing the data
    X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=test_size, random_state=42)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)
```

b) Data distribution and parameters initialization

Once the data is preprocessed, the master sends the data to every worker. After word, it initializes the parameters ($\boldsymbol{\beta}$).

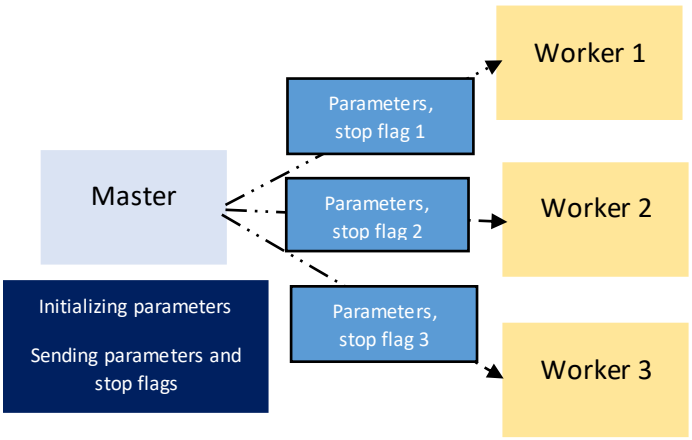


The code performing this step looks as follows.

```
frac_train = int(X_train.shape[0]/(num_workers-1))
frac_test = int(X_test.shape[0]/(num_workers-1))

#splitting data among workers
i = 0
for i in range(1, num_workers-1):
    data = {
        'X_train': X_train[((i-1)*frac_train):(i*frac_train),:],
        'X_test':  X_test[((i-1)*frac_test):(i*frac_test),:],
        'y_train': y_train[((i-1)*frac_train):(i*frac_train)],
        'y_test': y_test[((i-1)*frac_test):(i*frac_test)]
    }
    comm.send(data, dest=i)

data = {
    'X_train': X_train[(i*frac_train):,:],
    'y_train': y_train[(i*frac_train):],
    'X_test': X_test[(i*frac_test):,:],
    'y_test': y_test[(i*frac_test):]
}
comm.send(data, dest=(i+1))

n_samples = X.shape[0]
n_train_samples = X_train.shape[0]
n_test_samples = X_test.shape[0]

#initializing beta
beta = np.matrix(np.random.rand(X.shape[1],1))
```

c) Parameters and stop flag sharing

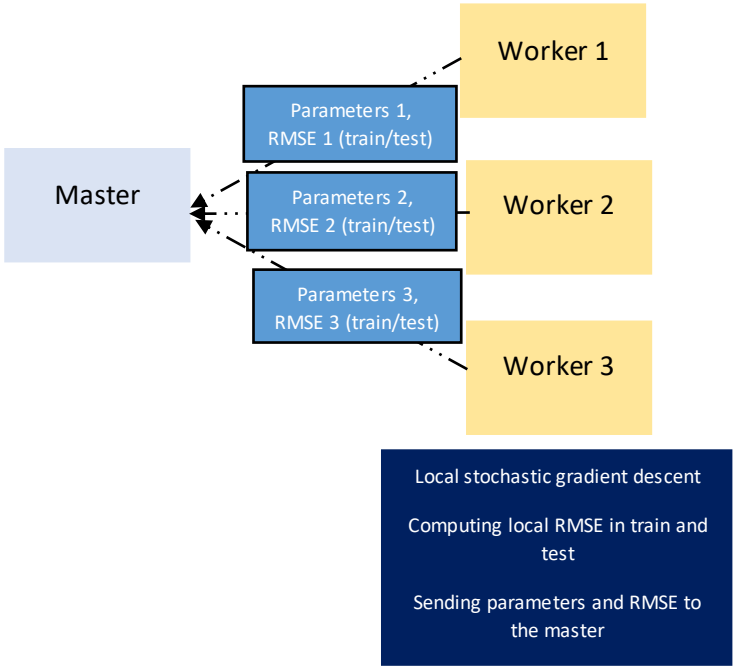
The **master** sends the parameters and stop flags (all initialized to *false* in the first iteration) to every **worker**, as a dictionary. In further iterations, the **master** also checks if the all the **workers** have *locally converged* or if they reach the maximum number of iterations. A worker is considered to have *locally converged* if the *local RMSE* difference between successive iterations is lower than a given threshold (or tolerance).




```
#sending parameters to the workers
for w in range(1, num_workers):
    data = {'beta':beta,
                'stop': stop}
    comm.send(data, dest=w)
```

d) Local stochastic gradient descent

Once every worker receives the parameters, it performs one epoch of stochastic gradient descent. The learning rate is a hyperparameter configured beforehand and it is the same for all of them. All the workers update the parameters by using the equation 1 and equation 2. When the computation is finished, the new parameters found by every worker $\boldsymbol{\beta}_j$ are sent to the master node.




```
        buff = comm.recv( source=0) #recigin information of current centers
and stop flag
        beta = buff["beta"]
        stop = buff["stop"]

        #performng stochastic gradient descent locally
        for j in range(n_train_samples):

            beta = beta - lr*grad(X_train[j,:], y_train[j,], beta)

        #computing train and test
        local_MSE_train = linear_loss (X_train, beta, y_train)
        local_MSE_test = linear_loss (X_test, beta, y_test)

        #creating dictionary to send to the master
        buff = {
            "local_beta": beta,
```

```
            "local_MSE_train": local_MSE_train,
            "local_MSE_test": local_MSE_test,
            "iter": i
        }

        comm.send(buff, dest=0)
```

The functions to compute the gradient and the RMSE (loss) are defined as follows.

```
def grad(X, y, beta):

    '''This function implements the gradient of the logistic loss'''

    grad=-X.T*(y-X*beta)
    return grad

def linear_loss (X, beta, y):

    '''Computes the loss of linear regression (MSE).
   The parameters are:
    - X is the matrix of features
    - beta is the vector of parameters for the linear regression
    - y is the target vector     '''

    y_pred = X*beta
    out = np.sum(np.array(y_pred-y)**2)

    return out
```

e) Parameters aggregation and convergence checking

After receiving the parameters, the master average their values by using the following equation:

$$\boldsymbol{\beta} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum \boldsymbol{\beta}_i$$

Where N is the number of workers. The **master** also checks if the stop conditions are met for every worker:

- Whether the maximum number of epochs was reached, or
- Whether the difference between the last and current RMSE falls below a threshold.

If one of this condition is hold for evry worker, the algorithm stops. Otherwise, the step **c** is repeated.

```
#receving and agreggating data from workers
        beta_ = np.matrix(np.zeros((X.shape[1],1)))


        for w in range(1, num_workers):

            #checking if the change in the MSE is small
            if(abs(MSE_train_diff[w]/n_train_samples)<tol):
                stop_worker[w]=True

            #receiving data from worker
            buff = comm.recv(source=w)

            #setting flag to stop worker if the limit of epochs is met
            if(buff["iter"]>epochs):
                stop_worker[w]=True

            #saving local RMSE
            local_MSE_train = buff["local_MSE_train"]
            MSE_train_diff[w] = MSE_train_list[w]-local_MSE_train
            MSE_train_list[w] = local_MSE_train
            MSE_test_list[w] = buff["local_MSE_test"]

            #aggregating beta
            beta_ += buff["local_beta"]

        #computing current RMSE
        current_RMSE_train = np.sqrt(sum(MSE_train_list)/n_train_samples)
        current_RMSE_test = np.sqrt(sum(MSE_test_list)/n_test_samples)

        #appending current RMSE
        RMSE_train_history.append(current_RMSE_train)
        RMSE_test_history.append(current_RMSE_test)

        #averaging beta
        beta = beta_/(num_workers-1)
```

experiments

May 17, 2019

1 Results

We run different experiments for different number of processes. We want to analyze the effect of the processes in the convergence of the algorithm and the RMSE. Moreover it is also interesting to analyze the time.

1.0.1 Virus Dataset (dataset 1)

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```
In [14]: !python sequential_version.py V

         name= "V"
         RMSE_train_sequential1 = np.load(name+"_RMSE_train.npy")
         RMSE_test_sequential1 = np.load(name+"_RMSE_test.npy")
         time_per_epoch_sequential1 = np.load(name+"_time.npy")
         acum_time_sequential1 = np.cumsum(time_per_epoch_sequential1)
         total_time_sequential1 = np.sum(time_per_epoch_sequential1)
         mean_time_sequential1 = np.mean(time_per_epoch_sequential1)
```

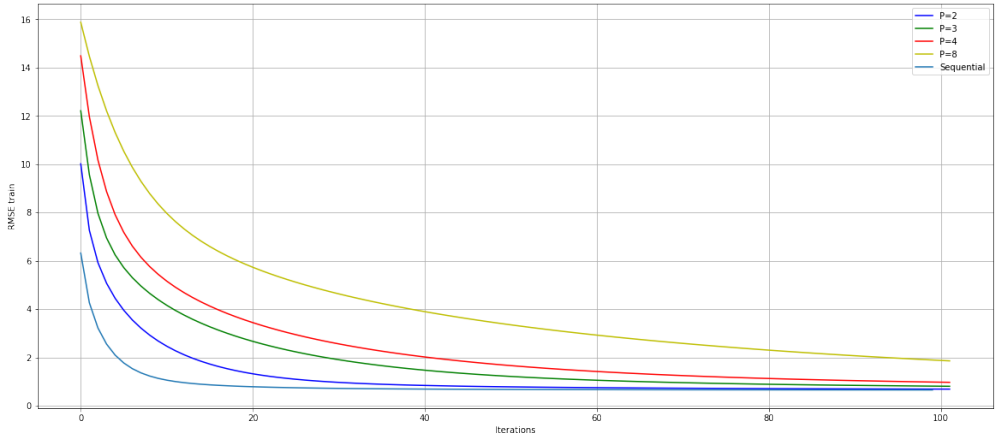
```
Loading virus dataset...
```

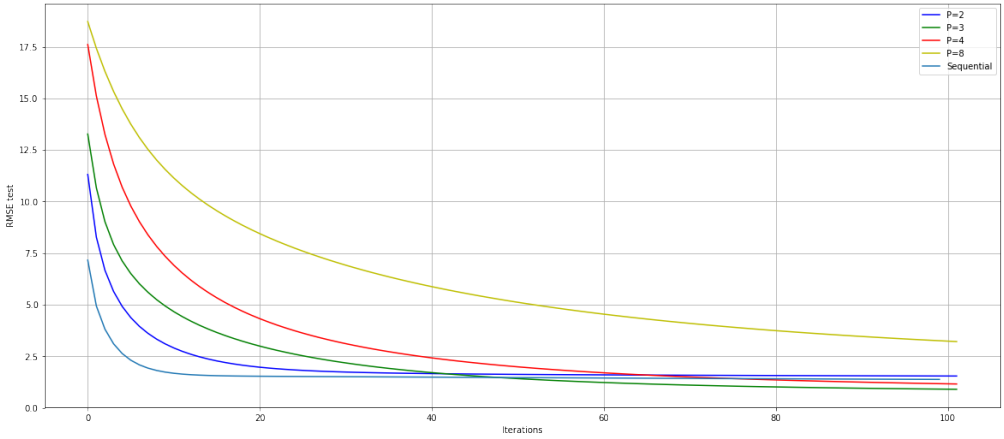
```
In [15]: fig1, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, ncols=1, figsize=(20,40)) # two ax
         col = ["b", "g", "r", "y"]
         processes = [2,3,4,8]

         RMSE_train_list1 = []
         RMSE_test_list1 = []
         acum_time_list1 = []
         total_time_list1 = []
         mean_time_list1 = []
```

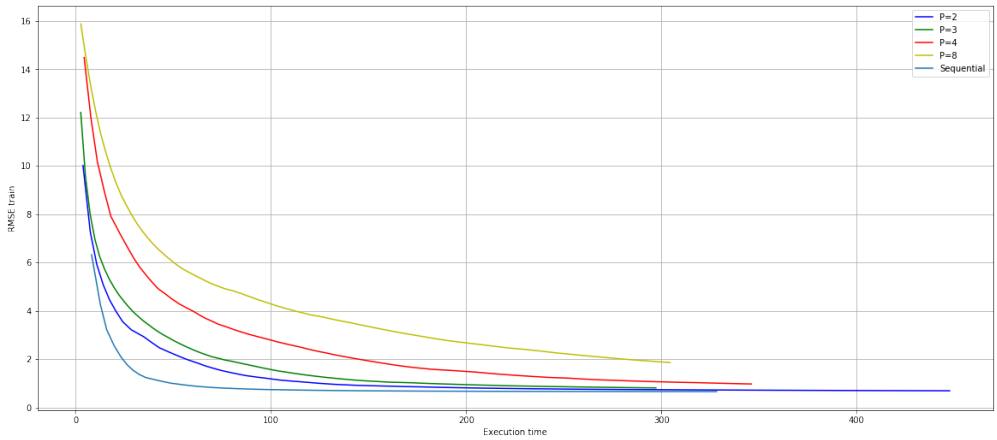
```
for i in range(len(processes)):

    p = processes[i]
    print("Running for ", p, " processes")

    name = "V"+str(p)
    !mpiexec -n $p python PSGD_MPI.py $name

    RMSE_train = np.load(name+"_RMSE_train.npy")
    RMSE_test = np.load(name+"_RMSE_test.npy")
    time_per_epoch = np.load(name+"_time.npy")
    acum_time = np.cumsum(time_per_epoch)
    total_time = np.sum(time_per_epoch)
    mean_time = np.mean(time_per_epoch)

    RMSE_train_list1.append(RMSE_train)
    RMSE_test_list1.append(RMSE_test)
    acum_time_list1.append(acum_time)
    total_time_list1.append(total_time)
    mean_time_list1.append(mean_time)

    ax1.plot(RMSE_train, col[i]+"-")
    ax2.plot(RMSE_test, col[i]+"-")
    ax3.plot(acum_time, RMSE_train, col[i]+"-")
    ax4.plot(acum_time, col[i]+"-")

ax1.plot(RMSE_train_sequential1)
ax2.plot(RMSE_test_sequential1)
ax3.plot(acum_time_sequential1, RMSE_train_sequential1)
ax4.plot(acum_time_sequential1)

ax1.set_xlabel("Iterations")
ax1.set_ylabel("RMSE train")
ax1.legend(("P=2","P=3","P=4","P=8","Sequential"))
ax1.grid()

ax2.set_xlabel("Iterations")
ax2.set_ylabel("RMSE test")
ax2.legend(("P=2","P=3","P=4","P=8","Sequential"))
ax2.grid()

ax3.set_xlabel("Execution time")
ax3.set_ylabel("RMSE train")
ax3.legend(("P=2","P=3","P=4","P=8","Sequential"))
ax3.grid()

ax4.set_xlabel("Execution time")
```

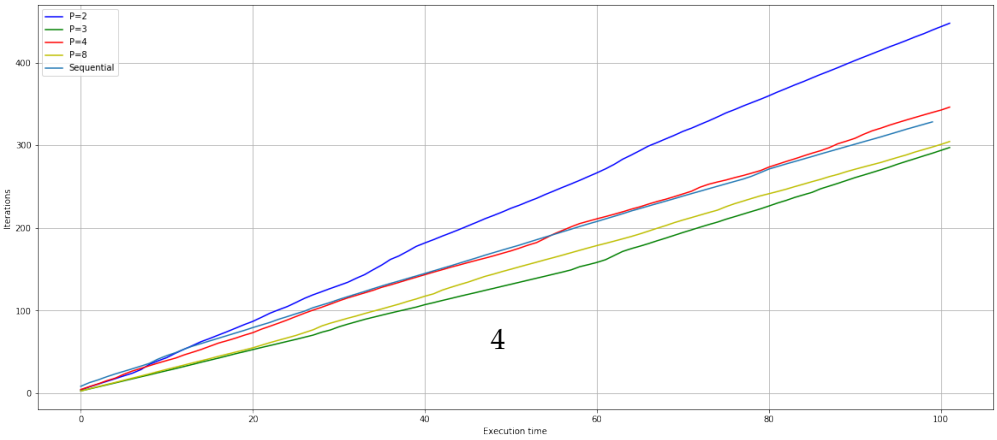
```
        ax4.set_ylabel("Iterations")
        ax4.legend(("P=2","P=3","P=4","P=8","Sequential"))
        ax4.grid()
```

```
Running for  2  processes
Loading virus dataset...
Reading and preprocessing time in master: 16.63631508119579
Running for  3  processes
Loading virus dataset...
Reading and preprocessing time in master: 18.087430093379226
Running for  4  processes
Loading virus dataset...
Reading and preprocessing time in master: 23.88613023601647
Running for  8  processes
Loading virus dataset...
Reading and preprocessing time in master: 21.70780289445247
```


P=2
P=3
P=4
P=8
Sequential
RMSE train
Iterations

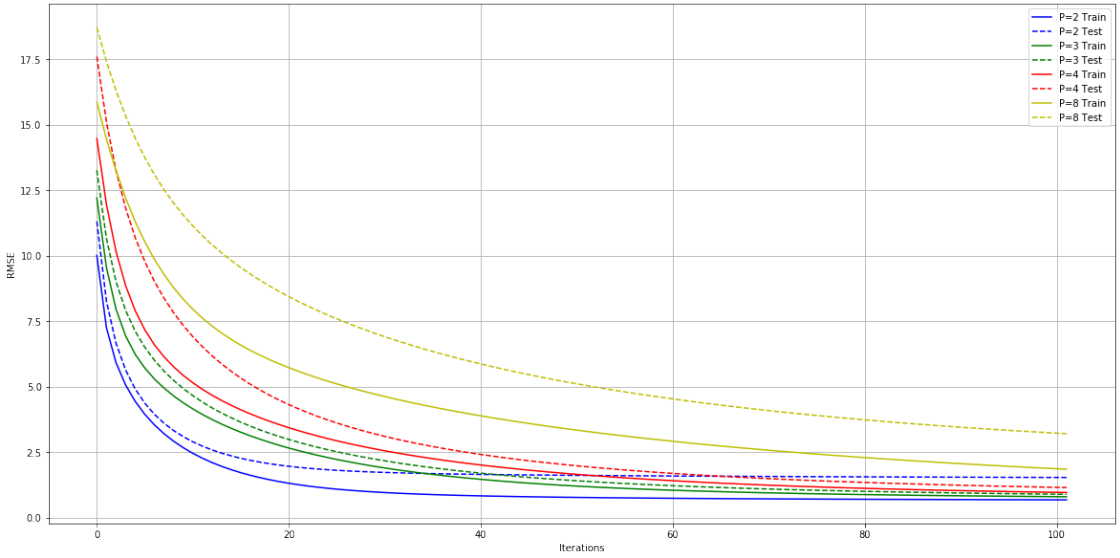

P=2
P=3
P=4
P=8
Sequential
RMSE test
Iterations


P=2
P=3
P=4
P=8
Sequential
RMSE train
Execution time


P=2
P=3
P=4
P=8
Sequential
Iterations
Execution time

In [26]:
```
fig1, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(20,10)) # two axes on figure


    for i in range(len(processes)):

        ax1.plot(RMSE_train_list1[i], col[i]+"-")
        ax1.plot(RMSE_test_list1[i], col[i]+"--")

    ax1.grid()
    ax1.set_xlabel("Iterations")
    ax1.set_ylabel("RMSE")
    ax1.legend(("P=2 Train","P=2 Test","P=3 Train","P=3 Test","P=4 Train","P=4 Test","P=8
```

Out[26]: <matplotlib.legend.Legend at 0x1446eada5f8>



1.0.2 KDD Cup Dataset (dataset 2)

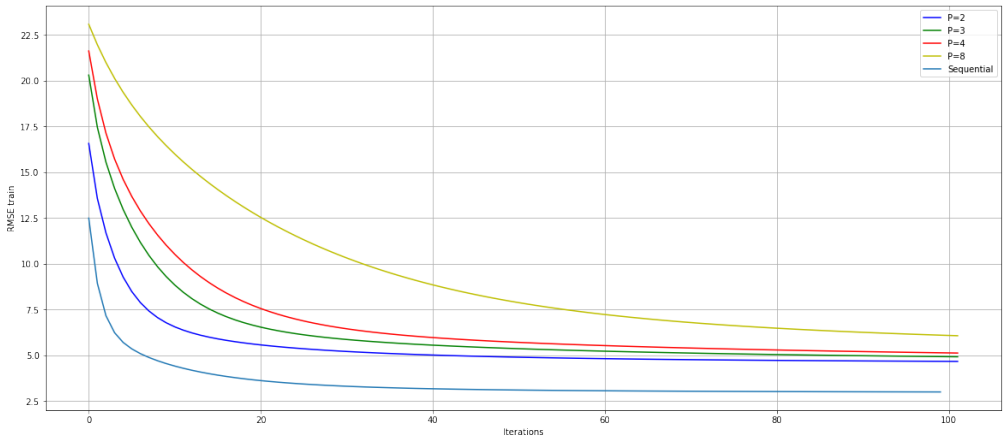
In [16]:
```
!python sequential_version.py K

    name = "K"
    RMSE_train_sequential2 = np.load(name+"_RMSE_train.npy")
    RMSE_test_sequential2 = np.load(name+"_RMSE_test.npy")
    time_per_epoch_sequential2 = np.load(name+"_time.npy")
    acum_time_sequential2 = np.cumsum(time_per_epoch_sequential2)
    total_time_sequential2 = np.sum(time_per_epoch_sequential2)
    mean_time_sequential2 = np.mean(time_per_epoch_sequential2)
```

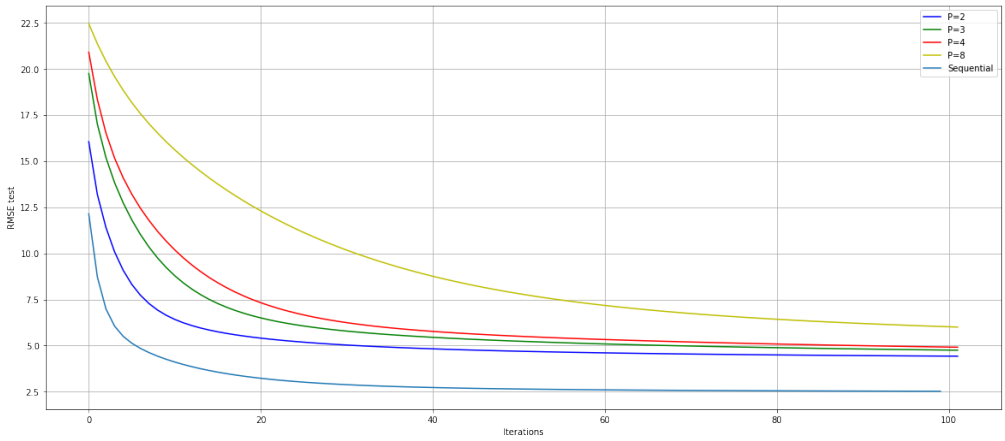
```
Loading KDD CUP dataset...


sys:1: DtypeWarning: Columns (8) have mixed types. Specify dtype option on import or set low_me
```

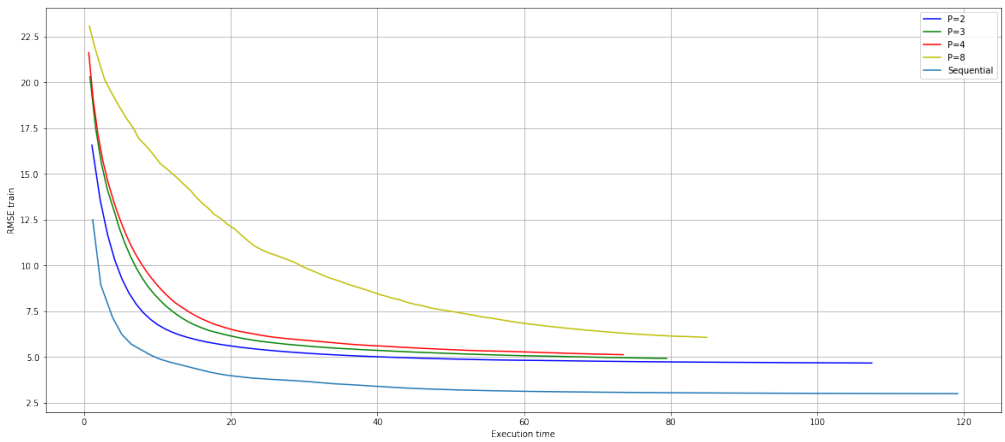
```
In [17]: fig1, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, ncols=1, figsize=(20,40)) # two ax

         RMSE_train_list2 = []
         RMSE_test_list2 = []
         acum_time_list2 = []
         total_time_list2 = []
         mean_time_list2 = []


         for i in range(len(processes)):

             p = processes[i]
             print("Running for ", p, " processes")

             name = "K"+str(p)
             !mpiexec -n $p python PSGD_MPI.py $name

             RMSE_train = np.load(name+"_RMSE_train.npy")
             RMSE_test = np.load(name+"_RMSE_test.npy")
             time_per_epoch = np.load(name+"_time.npy")
             acum_time = np.cumsum(time_per_epoch)
             total_time = np.sum(time_per_epoch)
             mean_time = np.mean(time_per_epoch)

             RMSE_train_list2.append(RMSE_train)
             RMSE_test_list2.append(RMSE_test)
             acum_time_list2.append(acum_time)
             total_time_list2.append(total_time)
             mean_time_list2.append(mean_time)

             ax1.plot(RMSE_train, col[i]+"-")
             ax2.plot(RMSE_test, col[i]+"-")
             ax3.plot(acum_time, RMSE_train, col[i]+"-")
             ax4.plot(acum_time, col[i]+"-")

         ax1.plot(RMSE_train_sequential2)
         ax2.plot(RMSE_test_sequential2)
         ax3.plot(acum_time_sequential2, RMSE_train_sequential2)
         ax4.plot(acum_time_sequential2)

         ax1.set_xlabel("Iterations")
         ax1.set_ylabel("RMSE train")
```

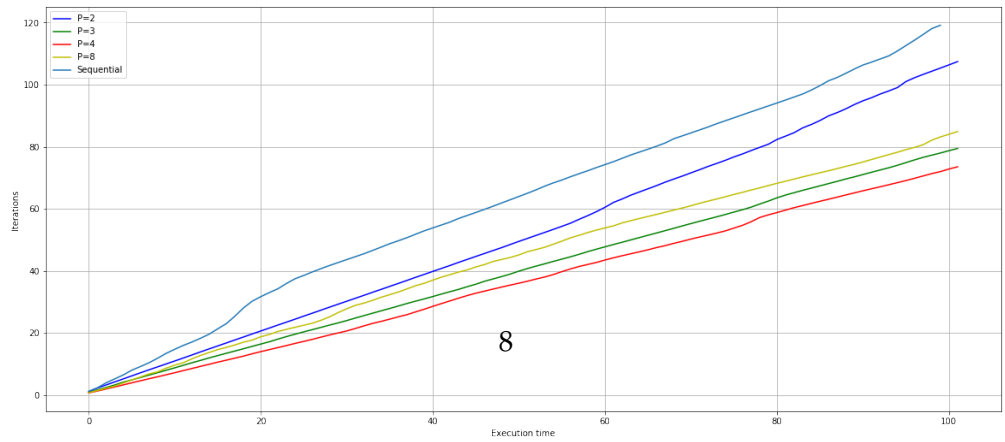
```
        ax1.legend(("P=2","P=3","P=4","P=8","Sequential"))
        ax1.grid()

        ax2.set_xlabel("Iterations")
        ax2.set_ylabel("RMSE test")
        ax2.legend(("P=2","P=3","P=4","P=8","Sequential"))
        ax2.grid()

        ax3.set_xlabel("Execution time")
        ax3.set_ylabel("RMSE train")
        ax3.legend(("P=2","P=3","P=4","P=8","Sequential"))
        ax3.grid()

        ax4.set_xlabel("Execution time")
        ax4.set_ylabel("Iterations")
        ax4.legend(("P=2","P=3","P=4","P=8","Sequential"))
        ax4.grid()
```

```
Running for  2  processes
Loading KDD CUP dataset...
Reading and preprocessing time in master: 8.841421584787895
Running for  3  processes


sys:1: DtypeWarning: Columns (8) have mixed types. Specify dtype option on import or set low_me


Loading KDD CUP dataset...
Reading and preprocessing time in master: 9.80353078970802
Running for  4  processes


sys:1: DtypeWarning: Columns (8) have mixed types. Specify dtype option on import or set low_me


Loading KDD CUP dataset...
Reading and preprocessing time in master: 8.881461834243964
Running for  8  processes


sys:1: DtypeWarning: Columns (8) have mixed types. Specify dtype option on import or set low_me


Loading KDD CUP dataset...
Reading and preprocessing time in master: 10.195448367507197


sys:1: DtypeWarning: Columns (8) have mixed types. Specify dtype option on import or set low_me
```

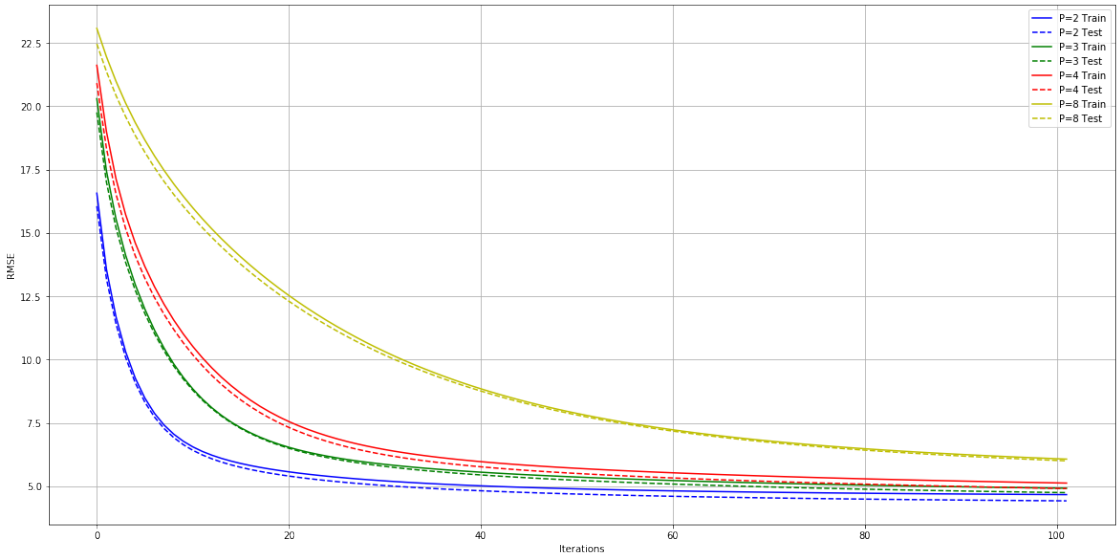

P=2
P=3
P=4
P=8
Sequential
RMSE train
Iterations


P=2
P=3
P=4
P=8
Sequential
RMSE test
Iterations


P=2
P=3
P=4
P=8
Sequential
RMSE train
Execution time


P=2
P=3
P=4
P=8
Sequential
Iterations
Execution time

```
In [27]: fig1, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(20,10)) # two axes on figure


         for i in range(len(processes)):

             ax1.plot(RMSE_train_list2[i], col[i]+"-")
             ax1.plot(RMSE_test_list2[i], col[i]+"--")

         ax1.grid()
         ax1.set_xlabel("Iterations")
         ax1.set_ylabel("RMSE")
         ax1.legend(("P=2 Train","P=2 Test","P=3 Train","P=3 Test","P=4 Train","P=4 Test","P=8

Out[27]: <matplotlib.legend.Legend at 0x1446e0876d8>
```



After running the experiment on both datasets, we can conclude that the distributed gradient descent may not give exactly the same final RMSE or convergence. Moreover, the more slaves (processes), the worst is the final RMSE (both in train and test). However, according to the plots, we can clearly see that there is an speed-up on the time used per iteration as the number of processes increases. The best number of processes to gain a speed-up in my personal laoptop is around 3-4 processes. While having many of them (e.g. 8), does not represent a comparable gain. In all the experiments, the worst time was certainly achieved with the sequential algorithm.

2 References

- Dataset 1: https://archive.ics.uci.edu/ml/datasets/Dynamic+Features+of+VirusShare+Executables

- Dataset 2: https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1998+Data
- Scikit learn: https://scikit-learn.org/stable/