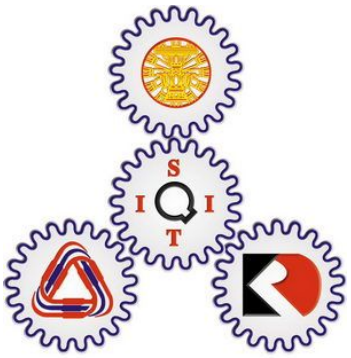


SCHOOL OF INFORMATION, COMPUTER AND COMMUNICATION TECHNOLOGY SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY THAMMASAT UNIVERSITY

CSS333 PARALLEL AND DISTRIBUTED COMPUTING

LINEAR REGRESSION PROJECT

To

Dr. Ekasit Kijsipongse

Members

Section 1

1. Ms. Wari	**Maroengsit**	**5822771333**
2. Ms. Natcha	**Sovipa**	**5822783346**
3. Mr. Dhawin	**Kritsernvong**	**5822791026**

Table of contents

Abstract

The purpose of this project is mainly to deep learning and studying about parallel computing and sequential computing, how parallel computing work, how it different from sequential computing, how better can it be, and how can it implement Linear Regression. Moreover, parallel computing also can reduce the computing time in the big dataset.

Introduction

In statistics, linear regression is a linear approach to modelling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression.

In this project, we implement only simple linear regression which is one of linear regression model with a single explanatory variable. It concerns two-dimensional sample points with one independent variable and one dependent variable (conventionally, the x and y coordinates in a Cartesian coordinate system) and finds a linear function (a non-vertical straight line) that, as accurately as possible, predicts the dependent variable values as a function of the independent variables. The adjective simple refers to the fact that the outcome variable is related to a single predictor.

So the reason we apply parallel computing to linear regression is for reducing the executing time and increasing the efficiency of computation. Since parallel programming saves time, allowing the execution of applications in a shorter wall-clock time. As a consequence of executing code efficiently, parallel programming often scales with the problem size, and thus can solve larger problems. In general, parallel programming is a means of providing concurrency, particularly performing simultaneously multiple actions at the same time.

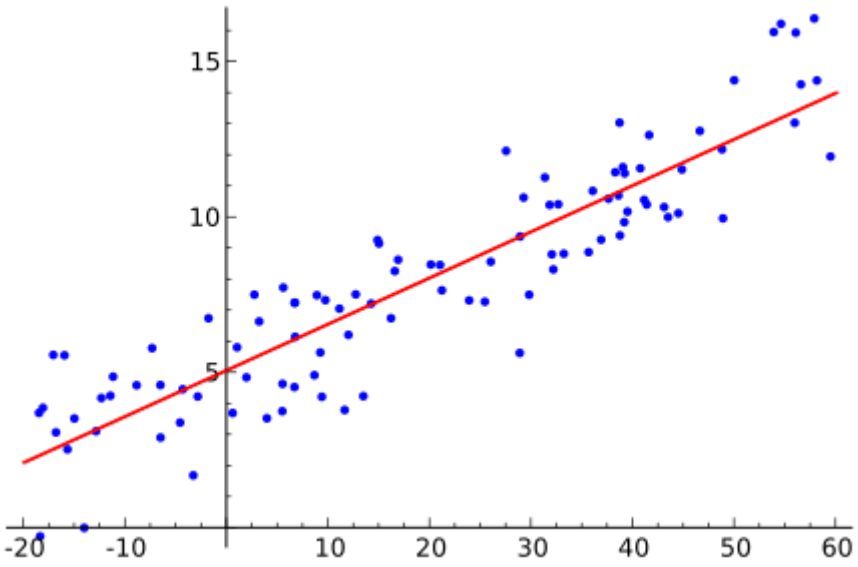


Figure 1 Linear regression analysis

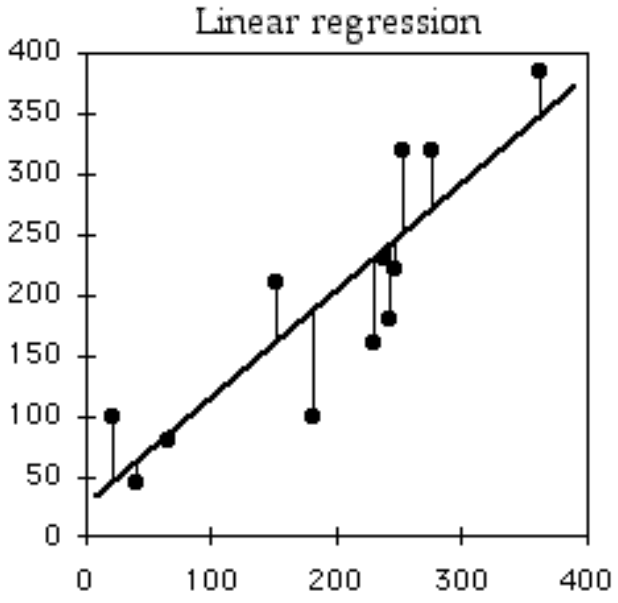


Figure 2 the data points (dots), linear regression line (thick line), and data points connected to the point on the regression line with the same X value (thin lines). The regression line is the line that minimizes the sum of the squared vertical distances between the points and the line.

Theorem

1. **Linear Regression**

Assume observations from a deterministic function with added Gaussian noise:

$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$ where $p(\epsilon|\beta) = \mathcal{N}(\epsilon|0, \beta^{-1})$

which is the same as saying,

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}).$$

Given observed inputs, $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and targets, $\mathbf{t} = [t_1, \ldots, t_N]^{\mathrm{T}}$, we obtain the likelihood function

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}).$$

Taking the logarithm, we get

$$\begin{aligned} \ln p(\mathbf{t}|\mathbf{w}, \beta) &= \sum_{n=1}^{N} \ln \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}) \\ &= \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi) - \beta E_D(\mathbf{w}) \end{aligned}$$

where $E_D(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2$ is the sum-of-squares error.

Computing the gradient and setting it to zero yields

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^{N} \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\} \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} = \mathbf{0}.$$

Solving for w, we get

$$\mathbf{w}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}$$

The Moore-Penrose pseudo $\boldsymbol{\Phi}^{\dagger}$

where

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}.$$

Maximizing with respect to the bias, w_0, alone, we see that

$$\begin{aligned} w_0 &= \bar{t} - \sum_{j=1}^{M-1} w_j \overline{\phi_j} \\ &= \frac{1}{N}\sum_{n=1}^{N} t_n - \sum_{j=1}^{M-1} w_j \frac{1}{N}\sum_{n=1}^{N} \phi_j(\mathbf{x}_n). \end{aligned}$$

We can also maximize with respect to β, giving

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{N}\sum_{n=1}^{N}\{t_n - \mathbf{w}_{\mathrm{ML}}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2$$

*Remark: A precision parameter β corresponding to the inverse variance of the distribution

2. Multiprocessing

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using sub-processes instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The multiprocessing module also introduces APIs which do not have analogs in the threading module. A prime example of this is the Pool object

which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism).

3. Pool

The Pool class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

4. Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Limitations of Linear Regression Model

1. Linear regression implements a statistical model that, when relationships between the independent variables and the dependent variable are almost linear, shows optimal results.
2. Linear regression is often inappropriately used to model non-linear relationships.
3. Linear regression is limited to predicting numeric output.
4. A lack of explanation about what has been learned can be a problem.

Advantages of Parallel Computing

1.Speed up.
2.Better cost per performance in the long run.

Disadvantages of Parallel Computing

1.Programming to target Parallel architecture is a bit difficult but with proper understanding and practice you are good to go.
2.Various code tweaking has to be performed for different target architectures for improved performance.
3.Communication of results might be a problem in certain cases.
4.Power consumption is huge by the multi core architectures.
and better cooling technologies are required in case of clusters.

How to use this project:

Requirement

1. Install Python 3.4 or more from https://www.python.org/downloads/
2. Require numpy and matplotlib library

Steps

1. Save the data that you want to compute in the filename “4-datapoint.txt”.
2. Open the code by python IDLE
3. Press f5 to run module.

Source Code

Sequential code

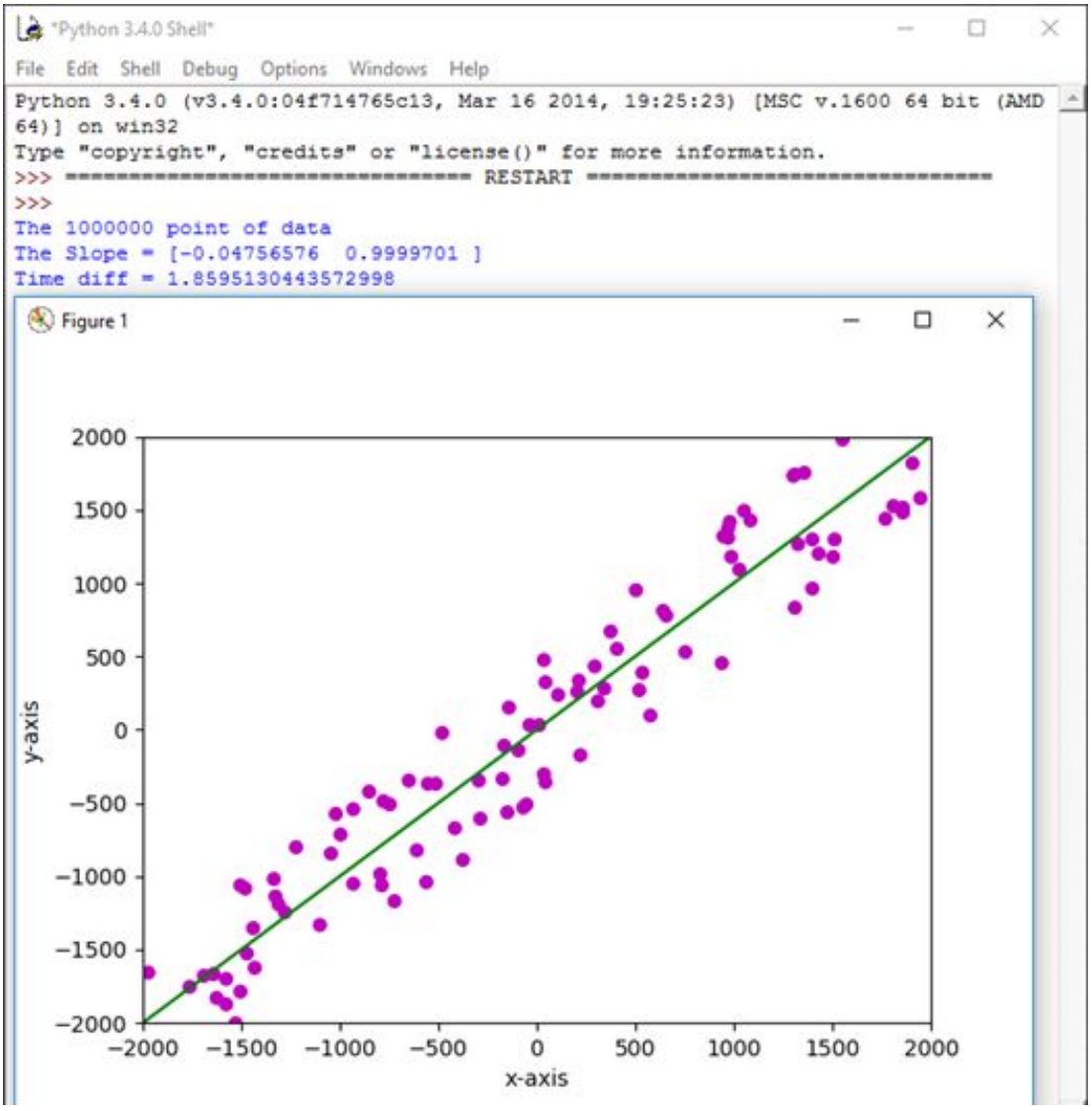
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import inv
import time
from random import randint

def tran(data):
    return [[row[i] for row in data] for i in range(len(data[0]))]
def multi(A,B):
    return [[sum(a * b for a, b in zip(A_row, B_col))
                        for B_col in zip(*B)]
                                for A_row in A]
def getData(num=4):
    filename=str(num)+"-datapoint.txt"
    text_file = open(filename, "r")
    X=[]
    Y=[]
    for line in text_file:
        x,y =line[:-1].split(",")
        X.append(int(x))
        Y.append(int(y))
    text_file.close()
    return X,Y

## Get Data
x,y1=getData(10000)
y=[]
y.append(y1)

start = time.time()

matXT=[[1]*len(x),x]
matX=tran(matXT)
matT=tran(y)
matXTT=np.matrix(multi(matXT,matT))
matXTX=np.matrix(multi(matXT,matX))
matW = inv(matXTX)
matW=matW*matXTT

listW = matW.tolist()
listW= [listW[0][0],listW[1][0]]

print("The Slope =",listW)
done = time.time()
## time exc
print("Time diff =",done-start)

## plot

plt.scatter(x[:100],y1[:100],color="m",marker="o",s=30)

x_pred=np.arange(-2000,2000,1)

plt.plot(x_pred,listW[1]*x_pred+listW[0],color='g')
##plt.plot([1,2,3,4],[1,4,9,16],'ro')
plt.axis([-2000,2000,-2000,2000])
plt.ylabel('y-axis')
plt.xlabel('x-axis')
plt.show()
```

Parallel code

```
from multiprocessing import Pool
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import inv
import time
from random import randint
def multiply(a):
    x=a[0]
    y=a[1]
    return sum(i[0] * i[1] for i in zip(x, y))
def getData(num=4):
    filename=str(num)+"-datapoint.txt"
    text_file = open(filename, "r")
    X=[]
    Y=[]
    for line in text_file:
        x,y =line[:-1].split(",")
        X.append(int(x))
        Y.append(int(y))
    text_file.close()
    return X,Y

if __name__ == '__main__':

##  GET DATA POINT
    x2,y=getData(10000)


    ## parallel compute
    start = time.time()

    x1=[1]*len(x2)
    multi = [[x1,x1],[x1,x2],[x2,x2],[x1,y],[x2,y]]
    pool = Pool(processes=4)
    ans=pool.map(multiply, multi)

    inver=inv(np.array([ans[:2],ans[1:3]]))
    temp=np.array(ans[3:5])
    listW=inver.dot(temp)

    print("The Slope =",listW)
    done = time.time()
    ## time exc
    print("Time diff =",done-start)

    ## plot

    plt.scatter(x2[:100],y[:100],color="m",marker="o",s=30)

    x_pred=np.arange(-2000,2000,1)

    plt.plot(x_pred,listW[1]*x_pred+listW[0],color='g')
    plt.axis([-2000,2000,-2000,2000])
    plt.ylabel('y-axis')
    plt.xlabel('x-axis')
    plt.show()
```

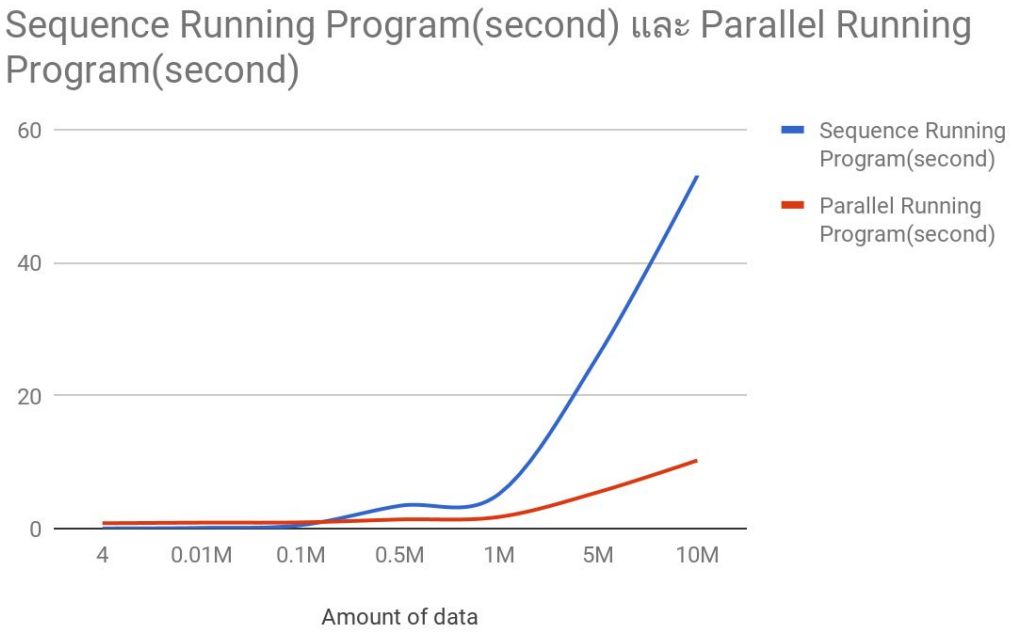
Result and Evaluation

According to the concept of parallel computing says that Speed up the execution time, increasing the efficiency of computation and better in cost per performance in the long run. The result show that parallel program works well in the big dataset while sequential good in small dataset.



Amount of data	Sequence Running Program(second)	Parallel Running Program(second)
4	0.02	0.8
0.01M	0.1	0.89
0.1M	0.51	0.93
0.5M	3.43	1.37
1M	5.21	1.75
5M	26.04	5.48
10M	53.1	10.25

*Remark : M represent million



Conclusion

We prove that parallel program better computation time in the big dataset while sequential good in small dataset , which combined with repetitive calculation, and could help reduce total time to get linear regression faster.That’s why parallel computing is very useful and faster to compute large data.