Final Project
Parallel Programming on Python Using joblib

Parallel and Distributed Programming, 2016 Fall
R05921075, 電機一, 鄭凱文

- Open source library: joblib

Joblib is a set of tools to provide lightweight pipelining in Python.
1. transparent disk-caching of the output values and lazy re-evaluation(memoize pattern)
2. easy simple **parallel computing**
3. logging and tracing of the execution
User documentation:http://pythonhosted.org/joblib
Source code:http://github.com/joblib/joblib

- Using joblib to parallel compute data normalization

The time-consuming and independant job: **Data normalization with matrix form**.
Data normalization is a common technique used in machine learning. With different data distribution, it is make sense to do normalization to make each data in gaussian distribution before training.
The formula is below,

$$x_{new} = \frac{x - \mu}{\sigma}$$

Where μ is mean of data, σ is standard derivation of data.
With this motivation, I write a non-parallel version python function code in the first.

```
def normalizeData(X):
    (p,q)=X.shape
    x_mean=np.zeros((p,1))
    x_sd=np.zeros((p,1))

    for i in range(p):
        for j in range(q):
            x_mean[i,0]=x_mean[i,0]+X[i,j]

    for i in range(p):
        x_mean[i,0]=x_mean[i,0]/q

    for i in range(p):
        for j in range(q):
            x_sd[i,0]=x_sd[i,0]+(X[i,j]-x_mean[i,0])*(X[i,j]-x_mean[i,0])

    for i in range(p):
        x_sd[i,0]=x_sd[i,0]/q
        x_sd[i,0]=sqrt(x_sd[i,0])

    for i in range(0,p):
        for j in range(0,q):
            X[i,j]=X[i,j]-x_mean[i,0]
            X[i,j]=X[i,j]/x_sd[i,0]

    return X, x_mean, x_sd
```

Figure 1, non-parallel version of data normalization function

We can find that the three red block is the main objective to be parallel.

- joblib API usage example

Non-parallel code

```
for i in range(10):
    sqrt(i**2)
```

Parallel by joblib

```
from joblib import Parallel, delayed
Parallel(n_jobs=2)(delayed(sqrt)(i**2) for i in range(10))
```

We can find that joblib need to use function call form to parallelization. Then, we can modify our code into function call form to fit joblib API.

- Parallel version of data normalization function

```
def normalizeDataParallel(X,core):
    (p,q)=X.shape                          ##non-parallel part
    x_mean=np.zeros((p,1))                 ##non-parallel part
    x_sd=np.zeros((p,1))                   ##non-parallel part

    x_mean=x_mean+Parallel(n_jobs=core)(           ##parallel part
        delayed(calmean)(i,X,p,q)                  ##parallel part
        for i in range(p))                         ##parallel part
    x_mean=np.diag(x_mean)             ##overhead
    x_mean.shape=(p,1)                 ##overhead

    for i in range(p):                     ##non-parallel part
        x_mean[i,0]=x_mean[i,0]/q          ##non-parallel part

    x_sd=x_sd+Parallel(n_jobs=core)(               ##parallel part
        delayed(calsd)(i,X,x_mean,p,q)             ##parallel part
        for i in range(p))                         ##parallel part
    x_sd=np.diag(x_sd)                 ##overhead
    x_sd.shape=(p,1)                   ##overhead

    for i in range(p):                     ##non-parallel part
        x_sd[i,0]=x_sd[i,0]/q              ##non-parallel part
        x_sd[i,0]=sqrt(x_sd[i,0])          ##non-parallel part

    tmp=Parallel(n_jobs=core)(                     ##parallel part
        delayed(processNormal)(i,X,x_mean,x_sd,p,q) ##parallel part
        for i in range(p))                         ##parallel part
    X=np.zeros((1,q))                  ##overhead
    for i in range(len(tmp)):          ##overhead
        X=np.row_stack((X,tmp[i]))     ##overhead
    X=X[1:,:]                          ##overhead

    return X, x_mean, x_sd
```

Figure 2, parallel version of data normalization function

```
def calmean(i,X,p,q):
    x_mean=np.zeros((p,1))                #overhead
    for j in range(q):
        x_mean[i,0]=x_mean[i,0]+X[i,j]
    return x_mean[:,0]

def calsd(i,X,x_mean,p,q):
    x_sd=np.zeros((p,1))                  #overhead
    for j in range(q):
        x_sd[i,0]=x_sd[i,0]+(X[i,j]-x_mean[i,0])*(X[i,j]-x_mean[i,0])
    return x_sd[:,0]

def processNormal(i,X,x_mean,x_sd,p,q):
    x_tmp=np.zeros((1,q))                 #overhead
    for j in range(0,q):
        x_tmp[0,j]=(X[i,j]-x_mean[i,0])/x_sd[i,0]
    return x_tmp
```

Figure 3, sub-function used in `normalizeDataParallel`

In code comment, we specified the parallel part, non-parallel part and overhead due to exclude race condition part.

- ## Performance measurement

OS: MacOS Yosemite
Hardware: Macbook Air 13" (early 2014)
Processor: 1.4 GHz Intel Core i5
Number of physical core: 2
Matrix size: (17, 92160), rows are variable, columns are data.

Using 2 physical core to parallel computing

	Non-Parallel execution time(sec)	Parallel execution time(sec)
test 1	9.984	5.838
test 2	9.643	5.932
test 3	9.396	5.860
Avg.	9.964	5.877

Speedup: **1.695**

- ## Open my source code on Github

https://github.com/kevin5566/Parallel_Normalization_joblib
There are three file: `pdpaux.py, main.py, data.csv`
"`pdpaux.py`" contain parallel and non-parallel data normalization function.
With any size of matrix, if each row is variable and column is data, then you can use this function to normalize data.
"`main.py`" is my testing code on my parallel function performance.
"`data.csv`" is the data I used.

```
import pdpaux
import numpy as np
import time

f=open('./data.csv','r')
f.readline()
X=pdpaux.readdata(f)          #(17, 5760)
f.close()
X=np.column_stack((X,X))
X=np.column_stack((X,X))
X=np.column_stack((X,X))
X=np.column_stack((X,X))
Y=X                          #(17, 92160)
(p,q)=X.shape
-----------------------------------------------------
print 'data dimension:\t',
print X.shape

t1=time.time()
(x1,x2,x3)=pdpaux.normalizeDataParallel(X,2)
t2=time.time()-t1

t3=time.time()
(y1,y2,y3)=pdpaux.normalizeData(Y)
t4=time.time()-t3

print 'Parallel execution time:\t',
print t2
print 'Non-Parallel execution time:\t',
print t4

print 'result checking:\t\t',
if np.asarray(np.where(x1-x2)).shape==(2, p*q):
    print 'pass'
else:
    print 'fail'
```

Figure 4, main.py

Above the green dash line, it is the read data part.
Below the green dash line, we use python library `time` to calculate runtime of two function.
In the last, we are interested in whether the parallel computing result is correct. Therefore, we check whether the difference of the two result is all zero. Finally, we find it is always pass.

● Conclusion

Nowaday, our personal computer is equipped with multicore. To learn how to make code parallel computing is a charming art for student in computer science.

In this final project, we use joblib API to parallelize our code and get a tremendous improvement. We all know the race condition is problem on parallel computing, so, except for the speedup, another issue should be noticed is whether the result is correct. We check the result to certify our code can get not only the speedup, but also correctness.

- Demo

Attach files are the same with Github version. Three file: `pdpaux.py, main.py, data.csv`
Before execution, your machine should install joblib. open terminal and type

```
sudo pip install joblib
```

After successful install, put the three file in the same directory, and type

```
python main.py
```

or `python -W ignore main.py` (to hide warning message)

Then, you can see result like this

```
data dimension: (17, 92160)
Parallel execution time:        5.584
Non-Parallel execution time:    9.983
result checking:                     pass
```

If you want to use your own data to test. You can modify the read data part of main.py. Use my data normalization API

```
pdpaux.normalizeDataParallel(X,2)
pdpaux.normalizeData(X)
```

to test performance.