Comparison of the performance of Machine Learning models executed sequentially and parallely

[1]Manas P Shankar, [2]Kaustubha DS, [3]Manish M, [4]Kaushik Kampli
(USN: [1]1MS18CS065,[2]1MS18CS058, [3]1MS18CS066, [4]1MS18CS057)

Abstract: Parallelism has been a really important factor in improvising the speed at which computations occur. Parallelism is achieved by running many threads at the same time across cores within a single node. This is an efficient strategy if the jobs do not have any kind of dependencies on each other since dependencies would force the computations to be executed in a serial manner. Our objective is to prove that parallelism when applied to various machine learning models would indeed lead to significant speedups that would reduce the time taken for computations. For this purpose, we have executed the codes for linear regression, KMeans clustering and Kernel density estimation both serially and in parallel. The percentage increase in the speedup values has shown that parallel execution is indeed advantageous.

1. Introduction

The application chosen for our work is the application of parallelism techniques in machine learning models, namely linear regression, KMeans clustering and Kernel density estimation. Through this experiment, we have tried to prove that parallel execution is faster than executing the code serially.

For the purpose of executing the code parallelly, the concept of threading was used. Thread context switching is very fast. Inside the processor, there are different contexts (program counter, general registers). Multithreading is not controlled by the operating system; rather, it is implemented at the firmware level.

To carry out parallelism directly, we make use of Numba, which is an open source Just-In-Time compiler, that translates a subset of Python and NumPy into fast machine code using LLVM, via the llvmlite Python package. It offers a range of options for parallelising Python code for CPUs and GPUs, often with only minor code changes.

We make use of the automatic parallelization features of jit() in numba, which attempts to identify operations in a user program that have parallel semantics. Instead of parallelizing individually, which would waste efficiency, it instead fuses adjacent operations together, to form one or more kernels that automatically run in parallel. The process is fully automated without modifications to the user program.

We also make use of the vectorize() function, where we manually create parallel kernels.

2. Method

Code for linear regression:

We first begin by importing all of the necessary libraries.

```
import numba
```

```python
import numpy as np
import argparse
import time
```

```python
run_parallel = numba.config.NUMBA_NUM_THREADS > 1

@numba.njit(parallel=run_parallel)
def linear_regression(Y, X, w, iterations, alphaN):
    for i in range(iterations):
        w -= alphaN * np.dot(X.T, np.dot(X,w)-Y)
    return w

def parallel_main():

    N = 20000
    D = 10
    p = 4
    iterations = 20
    alphaN = 0.01/N
    w = np.zeros((D,p))
    np.random.seed(0)
    points = np.random.random((N,D))
    labels = np.random.random((N,p))
    t1 = time.time()
    w = linear_regression(labels, points, w, iterations, alphaN)
    selftimed = time.time()-t1
    print("Bias values (parallel) : \n\n{}\n".format(w))
    print("Parallel Execution time (seconds) ", selftimed)
    print("checksum: ", np.sum(w),"\n\n")
    return np.sum(w), selftimed
```

```python
def linear_regression(Y, X, w, iterations, alphaN):
    for i in range(iterations):
        w -= alphaN * np.dot(X.T, np.dot(X,w)-Y)
    return w

def serial_main():
    N = 20000
    D = 10
    p = 4
    iterations = 20
    alphaN = 0.01/N
    w = np.zeros((D,p))
    np.random.seed(0)
    points = np.random.random((N,D))
    labels = np.random.random((N,p))

    print("Dataset size : {}".format(N))
    print("Dataset : \n",points[:10])

    t1 = time.time()
    w = linear_regression(labels, points, w, iterations, alphaN)
    selftimed = time.time()-t1
    print("\nBias values (serial) : \n\n{}\n".format(w))
    print("Serial Execution time (seconds) ", selftimed)
    print("checksum: ", np.sum(w))
    return np.sum(w), selftimed
```

```
checksum_s, stime = serial_main()
checksum_p, ptime = parallel_main()

print("Speedup :{:.2f}%".format((stime/ptime)*100))
if checksum_p == checksum_s:
    print("Successful Execution")
```

In the above code for Linear Regression, we highlight the functions serial_main() and parallel_main(), which effectively act the serial and parallel implementations of the function respectively. In both functions, we run the linear_regression() function for 20 iterations. In the linear regression function, we execute the standard regression formula Y_actual = (X^T.W), and get the Delta value by doing Y_expected - Y_actual, and finally modifying weights W using learning rate alphaN.

Code for KMeans clustering:

Import all of the necessary libraries.

```
import numba
import numpy as np
import math
from numba import jit, vectorize, njit, prange
import time
```

Below, we highlight the function used to execute **KMeans clustering in parallel mode**.

```
@njit(parallel=True)
def parallelized_kmeans(A, N, init_centroids, num_centers=5,
iterations=20, D=10):

    centroids = init_centroids
    for l in prange(iterations):

        dist =
np.array([[math.sqrt(np.sum((A[i,:]-centroids[j,:])**2)) for j in
range(num_centers)]for i in range(N)])

        labels = np.array([dist[i,:].argmin() for i in range(N)])

        centroids = np.array([[np.sum(A[labels==i,
j])/np.sum(labels==i) for j in range(D)]
                             for i in range(num_centers)])

    return centroids
```

Below, we highlight the function used to execute **KMeans clustering in serial mode**.

```
def serial_kmeans(A, N, init_centroids, clusters=5, iterations=20,
D=10):

    centroids = init_centroids
    for l in range(iterations):
        dist =
np.array([[math.sqrt(np.sum((A[i,:]-centroids[j,:])**2)) for j in
range(clusters)]for i in range(N)])
        labels = np.array([dist[i,:].argmin() for i in range(N)])
        centroids = np.array([[np.sum(A[labels==i,
j])/np.sum(labels==i) for j in prange(D)]
                             for i in prange(clusters)])

    return centroids
```

And finally, we highlight the function to **compare the running times of the code in serial and parallel execution modes**.

```
def compare_time():

    startp, starts, endp, ends = 0,0,0,0

    size = 10000
    features = 10
    clusters = 5
    iterations = 20

    np.random.seed(0)
    initial_centroids = np.random.ranf((clusters, features))
    print("Initial Centroids : ",initial_centroids)
    print("\n")
    data = np.random.ranf((size, features))

    startp=time.time()
    k_means = parallelized_kmeans(A=data, N=size,
init_centroids=initial_centroids)
    endp=time.time()

    print("Clusters (parallel) : \n",k_means)
    print("\nTime taken : {:.2f} seconds\n\n".format(endp-startp))

    starts=time.time()
    serial_k_means = serial_kmeans(A=data, N=size,
init_centroids=initial_centroids)
    endp=time.time()

    print("Clusters (serial) : \n",serial_k_means)
    print("\nTime taken : {:.2f} seconds\n".format(endp-startp))

    if serial_k_means.sum() == k_means.sum():
        print("\nSuccessful execution")
```

Calling the function to compare the running times of the execution of the code serially and parallelly.

```
compare_time()
```

Code for Kernel Density Estimation:

Import all of the necessary libraries.

```
import numpy as np
import pandas as pd
import numba
from numba import vectorize, jit
import time, random
```

Below we highlight the function to calculate the **Gaussian sum in KDE** using serial execution.

```
 def serial_kde(eval_points, samples, band):
    re_x = (eval_points[:, np.newaxis] - samples[np.newaxis, :]) /
band[np.newaxis, :]
    gaussian =
np.exp(-0.5*(re_x**2))/np.sqrt(2*np.pi)/band[np.newaxis, :]

    return gaussian.sum(axis=1)/len(samples)
```

Below is the function to return result of the Gaussian equation.

```
@jit(nopython=True)
def gaussian(x):

    return np.exp(-0.5*(x**2))/np.sqrt(2*np.pi)
```

```
@jit(nopython=True, parallel=True)
```

Below we highlight the function to calculate the **Gaussian sum in KDE** using parallel execution.

```
def parallel_kde(eval_points, samples, band):

    res = np.zeros_like(eval_points)
    for i in numba.prange(len(eval_points)):
        eval_x = eval_points[i]
        for s, b in zip(samples, band):
            res[i] += gaussian((eval_x-s)/b)/b
        res[i] /= len(samples)

    return res
```

```
def generate_input_samples():

    for dtype in [np.float64]:
        for n in [1000,5000]:
            sigma=0.5
            samples = np.random.normal(loc=0.0, scale=sigma,
size=n).astype(dtype)
            band = np.full_like(samples, 1.06*n**0.2*sigma)
            for n_eval in [10,1000, 5000]:
                cat = ('samples%d' %n,np.dtype(dtype).name)
                ep = np.random.normal(loc=0.0, scale=5.0,
size=n_eval).astype(dtype)
                yield dict(category=cat, x=n_eval, input_args=(ep,
samples, band), input_kwargs={})
```

```
val = generate_input_samples()
```

```
ip_args, ip_kwargs, size = tuple(),dict(),0
for item in val:
    ip_args = item['input_args']
    ip_kwargs = item['input_kwargs']
    size = item['x']
```

```
e_p, s, b = ip_args[0], ip_args[1], ip_args[2]
```

```
start_s = time.time()
serial_kde(e_p,s,b)
end_s = time.time()
```

```
print("Time taken for Serial implementation : {}
seconds".format(end_s-start_s))
print("Dataset Size : {} tuples".format(size))
```

```
start_p = time.time()
parallel_kde(e_p,s,b)
end_p = time.time()
```

Here, we compare the time taken for **parallel and serial executions**.

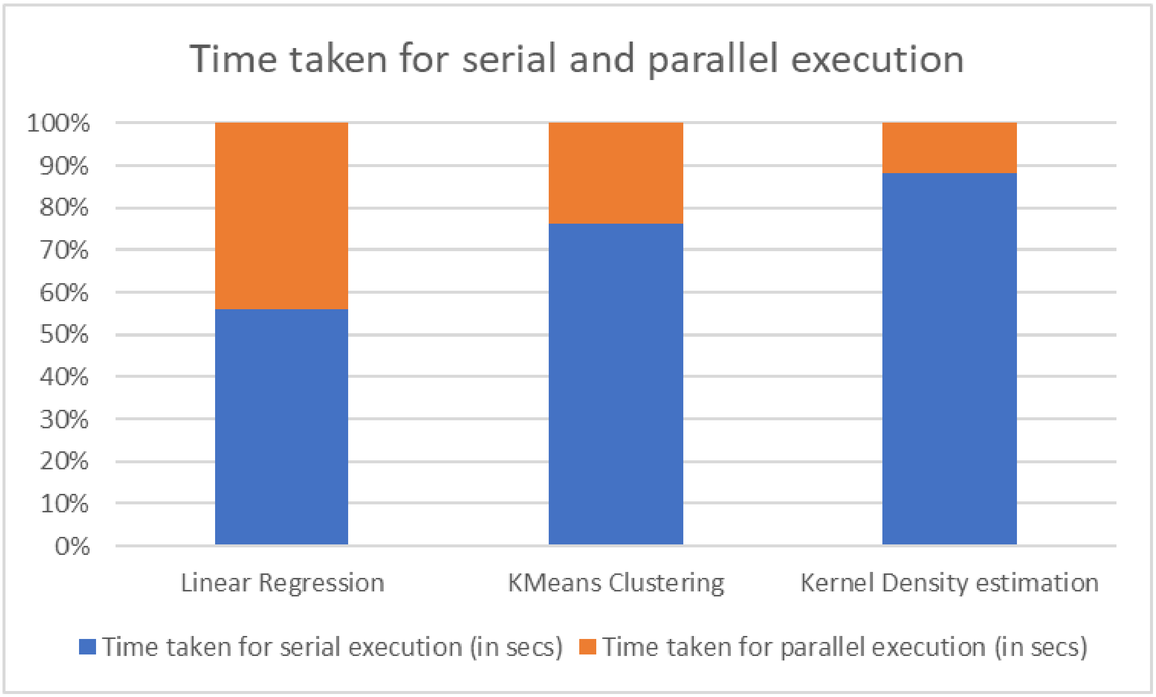
```
print("Time taken for Parallel implementation : {}
seconds".format(end_p-start_p))
print("Dataset Size : {} tuples".format(size))
```

```
print("Improvement with parallel implementation : {:.2f}
times".format(1/((end_p-start_p)/(end_s-start_s))))
```

3. Results

Machine learning model	Time taken for serial execution (in secs)	Time taken for parallel execution (in secs)	Percentage speedup due to parallelism
Linear Regression	0.033	0.026	128%
KMeans Clustering	10.19	3.17	321%
Kernel Density estimation	0.74	0.10	733%

The results clearly justify that parallelism leads to great speedups in processing compared to serial execution.



Limitation: We find that the percentage increase is not the same across machine learning models. Some models perform extremely well when executed parallely like Kernel Density estimation and KMeans clustering whereas some models do not give a good speedup like in linear regression.

References

[1] Zou, K. H., Tuncali, K., & Silverman, S. G. (2003). Correlation and simple linear regression. *Radiology*, *227*(3), 617-628.

[2] MacQueen, J. (1967, June). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (Vol. 1, No. 14, pp. 281-297).

[3] Terrell, G. R., & Scott, D. W. (1992). Variable kernel density estimation. *The Annals of Statistics*, 1236-1265.

[4] Banerjee, U., Eigenmann, R., Nicolau, A., & Padua, D. A. (1993). Automatic program parallelization. *Proceedings of the IEEE*, *81*(2), 211-243.

[5] Negri, A., Scannicchio, D. A., Touchard, F., & Vercesi, V. (2001). Multi thread programming.

[6] Lam, S. K., Pitrou, A., & Seibert, S. (2015, November). Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (pp. 1-6).

[7] Nicol, D., & Heidelberger, P. (1996). Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *6*(3), 210-242.