Homework2: Multicore Programming

Mingyi Xue*

May 20, 2018

1 Problem1 Multicore Programming

1.1 Parallel Algorithm

In KNN Algorithm, since each row of $\boldsymbol{X_{test}}$ could be independently computed to find its nearest neighbour in $\boldsymbol{X_{train}}$, the function to be parallelized can be simplified as follows,

Algorithm 1 Find The Nearest Neighbour For A Vector

Input: $\boldsymbol{X_{train}}$, $\boldsymbol{y_{train}}$, $\boldsymbol{x_{test}}$: one row in $\boldsymbol{X_{test}}$, y_{test}: label of the row
Output: $flag$

```
n ← 0
flag ← 0
index ← 0
N_train ← X_train.shape[0]
dist ← np.linalg.norm(X_train[0,:] − x_test)
for i = 1 → N_train do
    tmp ← np.linalg.norm(X_train[i,:] − x_test)
    if tmp < dist then
        index ← i
        dist ← tmp
    end if
end for
if y_test == y_train[index] then
    flag ← 1
end if
```

1.2 Parameters

Table 1: Default Parameters

name	value
$process\_num$	4
$dataset$	$data\_files.pl$

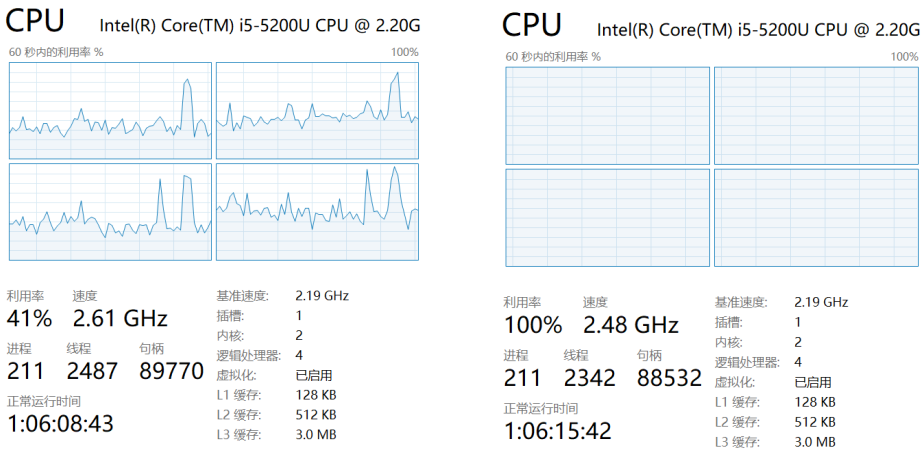
*GSP student from Nanjing University

1.3 Results

Table 2: Result of Parallel KNN Algorithm

Parallel method	time(sec)	accuracy(%)
single thread	188.1689	79.40
parallel by chunk	89.6257	79.40
parallel row by row	90.0971	79.40
parallel by chunk(not share memory)	97.5782	79.40
parallel row by row(not share memory)	95.5183	79.40

- Four processes cannot accelerate the program by four times.
- Parallelization by cutting $\boldsymbol{X_{test}}$ into four chunks or more extreme into rows costs similar time.
- Parallelization costs a little less time if treat $\boldsymbol{X_{train}}$ and $\boldsymbol{y_{train}}$ as global variables (or shared memory) than pass them as paramters to the function every time it is invoked.



(a) Single Thread (b) Parallelization

Figure 1: CPU Utilization of Parallel KNN

2 Problem2 Parallel Gradient Descent

2.1 Parallel Algorithm

Because the update of $\boldsymbol{\omega}$ relies on the result of former iteration, only the computation of the gradient descent can be parallelized. Cut $\boldsymbol{X_{train}}$ into four chunks:

Table 3: Chunks of Training Set

chunk
$\boldsymbol{X_{train}}[0 : size, :]$
$\boldsymbol{X_{train}}[size : 2 * size, :]$
$\boldsymbol{X_{train}}[2 * size : 3 * size, :]$
$\boldsymbol{X_{train}}[3 * size :, :]$

As a result, parallel gradient descent can be represented as follows,

$$\frac{\partial f(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}} = \sum_{i=1}^{N} \frac{-y_i}{1 + \mathrm{e}^{-y_i \boldsymbol{\omega}^T \boldsymbol{x_i}}} \boldsymbol{x_i} + \lambda \boldsymbol{\omega} \tag{1}$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{N_i} \frac{-y_j}{1 + \mathrm{e}^{-y_j \boldsymbol{\omega}^T \boldsymbol{x_j}}} \boldsymbol{x_j} + \lambda \boldsymbol{\omega} \tag{2}$$

2.2 Parameters

Table 4: Default parameters

name	value
n	4
N	$\boldsymbol{X_{train}}.shape[0]$
$size$	$np.ceil(N/n)$
$dataset$	$news20.binary.bz2$

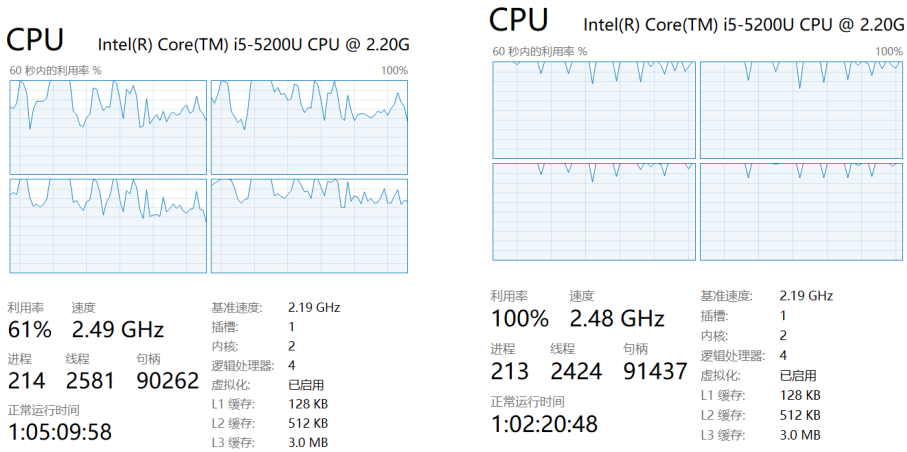
2.3 Results

Table 5: Result of Parallel Gradient Descent

Parallel method	time(sec)	training accuracy(%)	test accuracy(%)
single thread	764.1907	96.68	92.93
parallel by chunk	2420.9848	96.76	93.23

It is interesting to note that parallelized computation of sparse matrix costs even more time than the single thread version. One main reason is that modules like numpy and scipy have already optimized the computation of ndarray and matrix/sparse matrix, implemented by C++ or Fortran.

Though the utilization rate of CPU is higher than the single thread version if we explicitly use multiprocess in the code, extra CPU clock cycles will be consumed to allocate and free resources when calling mp.Pool.



(a) Single Thread (b) Parallelization

Figure 2: CPU Utilization of Parallel Gradient Descent