# Homework2: Multicore Programming

Mingyi Xue[*]

May 21, 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: $flag$
  $n \leftarrow 0$
  $flag \leftarrow 0$
  $index \leftarrow 0$
  $N_{train} \leftarrow \boldsymbol{X_{train}}.shape[0]$
  $dist \leftarrow np.linalg.norm(\boldsymbol{X_{train}}[0,:] - \boldsymbol{x_{test}})$
  for $i = 1 \rightarrow N_{train}$ do
      $tmp \leftarrow np.linalg.norm(\boldsymbol{X_{train}}[i,:] - \boldsymbol{x_{test}})$
      if $tmp < dist$ then
          $index \leftarrow i$
          $dist \leftarrow tmp$
      end if
  end for
  if $y_{test} == \boldsymbol{y_{train}}[index]$ then
      $flag \leftarrow 1$
  end if
```

### 1.2 Parameters

Table 1: Default Parameters

| name | value |
|---|---|
| *process_num* | 4 |
| *dataset* | *data_files.pl* |

[*]GSP student from Nanjing University

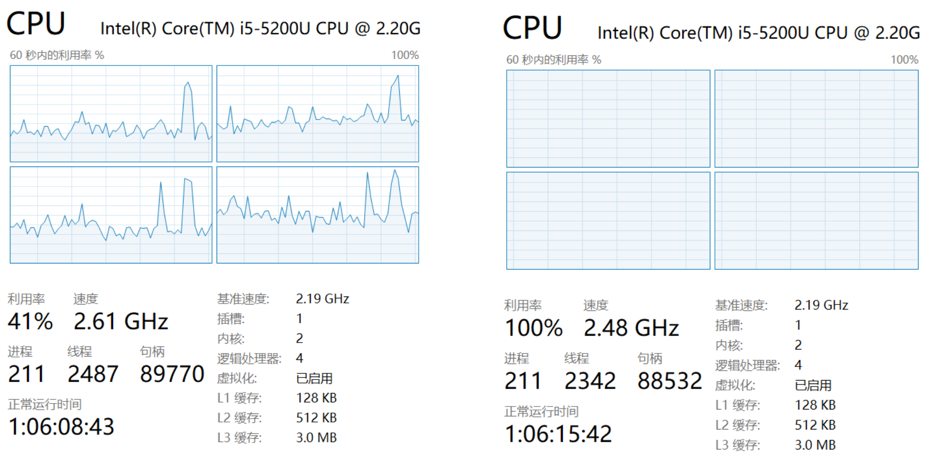
## 1.3 Results

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 |

## 1.4 Conclusion

- 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.



CPU Intel(R) Core(TM) i5-5200U CPU @ 2.20G
60 秒内的利用率 % 100%
利用率 41% 速度 2.61 GHz
进程 211 线程 2487 句柄 89770
正常运行时间 1:06:08:43

| | |
|---|---|
| 基准速度: | 2.19 GHz |
| 插槽: | 1 |
| 内核: | 2 |
| 逻辑处理器: | 4 |
| 虚拟化: | 已启用 |
| L1 缓存: | 128 KB |
| L2 缓存: | 512 KB |
| L3 缓存: | 3.0 MB |

(a) Single Thread

CPU Intel(R) Core(TM) i5-5200U CPU @ 2.20G
60 秒内的利用率 % 100%
利用率 100% 速度 2.48 GHz
进程 211 线程 2342 句柄 88532
正常运行时间 1:06:15:42

| | |
|---|---|
| 基准速度: | 2.19 GHz |
| 插槽: | 1 |
| 内核: | 2 |
| 逻辑处理器: | 4 |
| 虚拟化: | 已启用 |
| L1 缓存: | 128 KB |
| L2 缓存: | 512 KB |
| L3 缓存: | 3.0 MB |

(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,

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

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

In python, we can use numpy.ndarray broadcast to simplify and spped up calculation,

$$\begin{aligned} \boldsymbol{K} &= np.array(\boldsymbol{X} @ \boldsymbol{w}) * np.array(\boldsymbol{Y}) \\ \boldsymbol{M} &= -np.array(\boldsymbol{Y}) * \frac{1}{1 + \mathrm{e}^{\boldsymbol{K}}} \\ \frac{\partial f(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}} &= \boldsymbol{X^T M} + \lambda \boldsymbol{\omega} \end{aligned} \tag{3}$$

## 2.2 Results for data_files.pl

Table 4: Default parameters

| name | value |
|---|---|
| $n$ | 4 |
| $N_{train}$ | 10000 |
| $N_{test}$ | 1000 |
| $size$ | $np.ceil(N_{train}/n)$ |
| $dataset$ | $data\_files.pl$ |

Table 5: Result of Parallel Gradient Descent

| Parallel method | time($sec$) | training accuracy(%) | test accuracy(%) |
|---|---|---|---|
| single thread | 1.7140 | 76.21 | 75.30 |
| parallel by chunk | 31.5040 | 76.21 | 75.30 |

## 2.3 Results for news20.binary.bz2

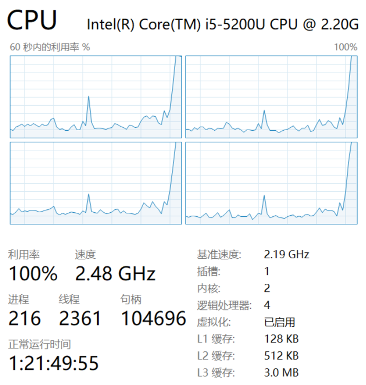
Table 6: Default parameters

| name | value |
|---|---|
| $n$ | 4 |
| $N$ | $\boldsymbol{X_{train}}.shape[0]$ |
| $split\_percent$ | 0.2 |
| $size$ | $np.ceil(N/n)$ |
| $dataset$ | $news20.binary.bz2$ |

Table 7: 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 |

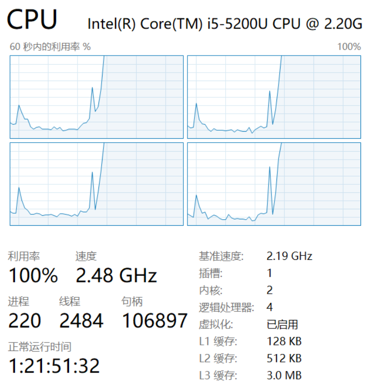
## 2.4 Conclusion



```
利用率    速度
100%     2.48 GHz
进程   线程   句柄
216   2361   104696
正常运行时间
1:21:49:55

基准速度:     2.19 GHz
插槽:         1
内核:         2
逻辑处理器:   4
虚拟化:       已启用
L1 缓存:      128 KB
L2 缓存:      512 KB
L3 缓存:      3.0 MB
```

(a) Single Thread

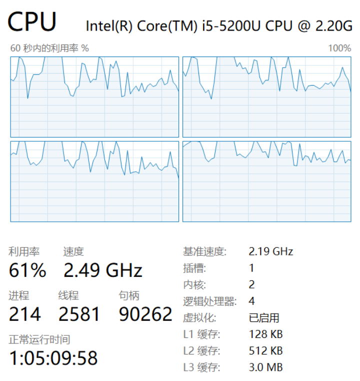


```
利用率    速度
100%     2.48 GHz
进程   线程   句柄
220   2484   106897
正常运行时间
1:21:51:32

基准速度:     2.19 GHz
插槽:         1
内核:         2
逻辑处理器:   4
虚拟化:       已启用
L1 缓存:      128 KB
L2 缓存:      512 KB
L3 缓存:      3.0 MB
```

(b) Parallelization

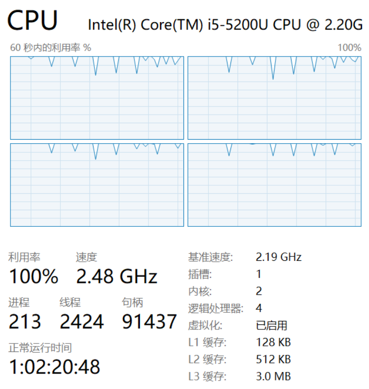
Figure 2: CPU Utilization of Parallel Gradient Descent for data_files.pl



| 利用率 | 速度 | |
|---|---|---|
| 61% | 2.49 GHz | |
| 进程 | 线程 | 句柄 |
| 214 | 2581 | 90262 |
| 正常运行时间 | | |
| 1:05:09:58 | | |

| | |
|---|---|
| 基准速度: | 2.19 GHz |
| 插槽: | 1 |
| 内核: | 2 |
| 逻辑处理器: | 4 |
| 虚拟化: | 已启用 |
| L1 缓存: | 128 KB |
| L2 缓存: | 512 KB |
| L3 缓存: | 3.0 MB |

(a) Single Thread



| 利用率 | 速度 | |
|---|---|---|
| 100% | 2.48 GHz | |
| 进程 | 线程 | 句柄 |
| 213 | 2424 | 91437 |
| 正常运行时间 | | |
| 1:02:20:48 | | |

| | |
|---|---|
| 基准速度: | 2.19 GHz |
| 插槽: | 1 |
| 内核: | 2 |
| 逻辑处理器: | 4 |
| 虚拟化: | 已启用 |
| L1 缓存: | 128 KB |
| L2 缓存: | 512 KB |
| L3 缓存: | 3.0 MB |

(b) Parallelization

Figure 3: CPU Utilization of Parallel Gradient Descent for news20.binary.bz2

It is interesting to note that parallelized computation of 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. If gradient descent algorithm is implemented with vectorization, the program is parallel implicitly.

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.