Report K-means clustering algorithm

台灣科技大學: 陳勁銘

StudentID: M10903311

實驗室:CFDLAB

前置作業

Ref

- Ref01
- Ref002
- GitHub
- 老師上課講義

To do list

- [√] All the needed data by the k-means algorithm should be stored in struct myData declared in kmeans.hpp
- [√] OpenMP
- [√] Binding

資料整理

分類

- 非監督式學習
- 物以類聚

過程

1. 我們先設定好要分成多少(k)群。

2. 然後在feature space(x軸身高和y軸體重組出來的2維空間，假設資料是d維，則會組出d維空間)隨機給k個群心
3. 每個資料都會所有k個群心算歐式距離(歐基李德距離Euclidean distance，其實就是直線距離公式，從小學到大的那個距離公式，這邊距離當然也可以換成別種距離公式，但基本上都還是以歐式距離為主)
4. 將每筆資料分類判給距離最近的那個群心。
5. 每個群心內都會有被分類過來的資料，用這些資料更新一次新的群心。
6. 一直重複3–5，直到所有群心不在有太大的變動(收斂)，結束。

Discussion01 : Which data layout is faster? AOS or SOA?

> Back up your discussions by conducting experiments by varying DIM, N, and K

data struct

```
struct myData {
    size_t DIM; // number of dimention
    size_t N; //number of point
    size_t K; // number of k clusters
    vector<float> pts; // 數量為 DIM * N
    vector<float> currentCentroids; // 數量為 k * DIM
    vector<float> oldCentroids; // 數量為 k * DIM
    vector<size_t> group; // 數量為K
};
```

主程式中:

```
myData data;//宣告一個 data 的 data struct
```

- 注意事項
  - currentCentroids 如果K在外圈 >> DIM要連續

main loop

我們最後輸出

```
    std::cout   << "\n"
                << argv[1] << ", "
```

```
                << argv[2] << ", "
                << it << ","
                << timer[0].elapsedTime() << ","
                << timer[1].elapsedTime() << ","
                << timer[2].elapsedTime() << endl;
```

讀資料區 readCSV

討論 : 所有的vector使用at存取資料使得資料更為安全

1. AOS: coordinates of the same point are stored continuously.
    - x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4, …

kmeansAOS.cpp

```
    inp.clear();
    inp.seekg(0, inp.beg);
    data.pts.resize(noLines*noAttributes);
    for (size_t whichPt = 0; whichPt < noLines; ++whichPt) {
        inp.getline(buf, 4096);
        auto it = buf;
        for (size_t whichDim = 0; whichDim < noAttributes;++whichDim) {
            auto x = atof(it);
            data.pts[whichPt*noAttributes+whichDim] = x;            ///AOS
            while(*it!=',') ++it;
            it++;
        }
    }
```

2. SOA: coordinates of the same dimension are stored continuously.
    - x1, x2, x3, x4, … y1, y2, y3, y4, … z1, z2, z3, z4, …
      kmensSOA.cpp

```
    inp.clear();
    inp.seekg(0, inp.beg);
    data.pts.resize(noLines*noAttributes);
    for (size_t whichPt = 0; whichPt < noLines; ++whichPt) {
        inp.getline(buf, 4096);
        auto it = buf;
        for (size_t whichDim = 0; whichDim < noAttributes;++whichDim) {
            auto x = atof(it);
            data.pts[whichPt+whichDim*noLines] = x;                ///SOA
```

```
            while(*it!=',') ++it;
            it++;
        }
    }
```

- 輸入中 外圈為 `whichPt` 內圈為 `whichDim`
  - `AOS` 友善
  - `SOA` 不友善

`UpdateCentroids` 中

> 我們決定了 `currentCentroids` 與 `oldCentroids` 的矩陣排法，可以觀察到外迴圈應該要放 `count_k` 而 內迴圈則是 `whichDim`

```
data.currentCentroids.resize(data.K*data.DIM);      // vector 大小
data.currentCentroids[data.DIM*count_k+whichDim]    // 輸入資料
data.currentCentroids.at(data.DIM*count_k+whichDim) // 取出資料
```

```
data.oldCentroids.resize(data.K*data.DIM);       // Vector 大小
data.oldCentroids[count_k+whichDim*data.K]       // 輸入資料
data.oldCentroids.at(count_k+whichDim*data.K)    // 取出資料
```

AssignGroups(迭帶區)

AssignGroups_part01

以下這行可以省略，因為我們只需要比較大小，而**不需要知道真正的數值**

```
distance_sum_DIM = sqrt(distance_sum_DIM);
```

- 取點
  - 看到再取點時 外迴圈為 `whichPt` ， 內迴圈才是 `whichDim`
  - AOS 不友善 `data.pts.at(whichPt+whichDim*data.N)`
  - SOA 友善 `data.pts.at(whichPt*data.DIM+whichDim)`
- 寫入 :
  - 寫入情況下，我們知道在C++中矩陣如果為 `dis[whichPt][count_k]` 則count為連續則要放內圈
  - 在這個迴圈中 `SOA` 與 `AOS` **都是優良的**

SOA

```
for (size_t whichPt = 0; whichPt < N_value; whichPt++)
{
    for (size_t count_k = 0; count_k < K_value; count_k++)
    {
        distance_sum_DIM = 0;
        for (size_t whichDim = 0; whichDim < DIM_value ; whichDim++)
        {
            distance_sum_DIM += pow((data.currentCentroids.at(data.DIM*count_k \
                        +whichDim)- data.pts.at(whichPt+whichDim*data.N)),2);
        }
        // distance_sum_DIM = sqrt(distance_sum_DIM);  // 刻意省略
        dis[whichPt][count_k] = distance_sum_DIM;
    }
}
```

AOS

```
for (size_t whichPt = 0; whichPt < N_value; whichPt++)
{
    for (size_t count_k = 0; count_k < K_value; count_k++)
    {
        distance_sum_DIM = 0;
        for (size_t whichDim = 0; whichDim < DIM_value ; whichDim++)
        {
            distance_sum_DIM += pow((data.currentCentroids.at(data.DIM*count_k \
                       +whichDim)- data.pts.at(whichPt*data.DIM+whichDim)),2);
        }
        // distance_sum_DIM = sqrt(distance_sum_DIM); // 刻意省略
        dis[whichPt][count_k] = distance_sum_DIM;
    }
}
```

AssignGroups_part02

這裡也沒什麼問題，都為連續只不過之後要注意 `dis_min` 在平行化的情況下要設立私有變數

AOS 與 AOS

```
    for (size_t whichPt = 0; whichPt < N_value; whichPt++)
    {
    dis_min = FLT_MAX;
```

```
    for (size_t count_k = 0; count_k < K_value; count_k++)
    {
        if (dis_min > dis[whichPt][count_k] )
        {
            dis_min = dis[whichPt][count_k];
            data.group[whichPt] = count_k;
        }
        }
    }
```

UpdateCentroids

- 取點
  - 看到再取點時 內迴圈為 `whichPt` , 外迴圈才是 `whichDim`
  - AOS 友善 `data.pts.at(whichPt+whichDim*data.N)`
  - SOA 不友善 `data.pts.at(whichPt*data.DIM+whichDim)`
- 寫入
  - `data.oldCentroids` 寫入檔案也是友善(`count_k` 在內迴圈)

`SOA`

```
int counter_tem= 0;
float center_tem = 0.;
data.oldCentroids.resize(K_value*DIM_value);
for (size_t whichDim = 0; whichDim < data.DIM; whichDim++)
{
    for (size_t count_k = 0; count_k < K_value; count_k++)
    {
        center_tem = 0.;
        counter_tem = 0;
        for (size_t whichPt = 0; whichPt < N_value; whichPt++)
        {
            if (data.group[whichPt] == count_k)
            {
                center_tem+=data.pts.at(whichPt+whichDim*data.N);
                counter_tem++;
            }
        }
        data.oldCentroids[count_k+whichDim*data.K]=\
                        center_tem/float(counter_tem);
    }
}
```

AOS

```
    int counter_tem= 0;
    float center_tem = {0.};
    data.oldCentroids.resize(K_value*DIM_value);
    for (size_t whichDim = 0; whichDim < data.DIM; whichDim++)
    {
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
            center_tem = 0.;
            counter_tem = 0;
            for (size_t whichPt = 0; whichPt < N_value; whichPt++)
            {
                if (data.group[whichPt] == count_k)
                {
                    center_tem+=data.pts.at(whichPt*data.DIM+whichDim);
                    counter_tem++;
                }
            }
            data.oldCentroids[count_k+whichDim*data.K]= \
                            center_tem/float(counter_tem);
        }
    }
```

檢查收斂 HasConverged

主迴圈中每一次也都有檢查所以我們也進行分析

- 寫入
- 讀值相當友善

AOS 與 SOA

```
    for (size_t whichDim = 0 ; whichDim < DIM_value ; whichDim++)
    {
        for (size_t count_k  = 0 ; count_k < K_value ; count_k++)
        {
            tolerance_sum =+ std::abs(data.oldCentroids.at(count_k+whichDim*data.K) - data.cu
        }
    }

    for (size_t i =0; i <K_value*DIM_value ; i++ )
    {
```

```
        data.currentCentroids[i] = data.oldCentroids.at(i);
    }
```

觀察到的有趣資訊

1. `return 255;` 就是 `return -1;`

2. `#pragma once` 可以取代以前 `#if define....`

3. #include 為C++ 11的時間庫，提供計時，時鐘等功能。

加入OpenMP

有加入Fisrt Touch 的參數

1. data.pts // 刻意加入
2. dis // 自然加入

還沒

1.

readCSV(平行前後coding比較)

Part2 加入，First Touch

> 我們必須要想辦法在資料一開始就被安排在各個 thread 上面然後加入 binding 試看看， 因為等等讀數值的時候沒有平行

```
#pragma omp parallel for schedule(static)
for (size_t whichPt = 0; whichPt < noLines; ++whichPt) {
{
    for (size_t whichDim = 0; whichDim < noAttributes;++whichDim) {
        data.pts[whichPt*noAttributes+whichDim] = 0.0;
    }
}
```

Part3 真正的讀數值

> 目前還沒計畫 IO區塊平行 這邊可能會因為上一個 Part2 中我們設計的記憶體關係，造成 Part3 比較慢，**值得套論兩者關係**

```
for (size_t whichPt = 0; whichPt < noLines; ++whichPt) {
    inp.getline(buf, 4096);
    auto it = buf;
    for (size_t whichDim = 0; whichDim < noAttributes;++whichDim) {
        auto x = atof(it);
        data.pts[whichPt*noAttributes+whichDim] = x;
        while(*it!=',') ++it;
        it++;
    }
}
```

InitializeCentroid

主要目標也是要讓疊帶的時候 First private

那就先跳過了

AssignGroups(迭代區)

Part1

```
// 平行前
    for (size_t whichPt = 0; whichPt < N_value; whichPt++)
    {
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
            distance_sum_DIM = 0;
            for (size_t whichDim = 0; whichDim < DIM_value ; whichDim++)
            {
                distance_sum_DIM += pow((data.currentCentroids.at(count_k+whichDim*data.K)- c
            }
            // distance_sum_DIM = sqrt(distance_sum_DIM);
            dis[whichPt][count_k] = distance_sum_DIM;
        }
    }

    for (size_t whichPt = 0; whichPt < N_value; whichPt++)
    {
    dis_min = FLT_MAX;
```

```
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
        if (dis_min > dis[whichPt][count_k] )
        {
            dis_min = dis[whichPt][count_k];
            data.group[whichPt] = count_k;
        }
        }
    }
```

這裡在做的事情其實很簡單，就是把每一個**獨立點**與現**有的形心距離**算出,存入矩陣 `dis[whichPt][count_k]`

所以我們可以很清楚的資料我們可以平行**各個點**，而形心(`data.currentCentroids`)為 唯獨入資料根據點來平行，整體程式是用 whichPT來平行所以還在思考要如何在這個大迴圈中一直需要的數值 `data.currentCentroids` 在各個thread中的記憶體，但因為大小很小所以可能已經可以在 cacheLine 裡面了

- 而這邊的一次會跑的迴圈大小為 DIM_value * K_value * N_value(可以平行) 可以說是最多計算的區域又有pow這個function
  - distance_sum_DIM += pow((data.currentCentroids.at(count_k+whichDim*data.K)-data.pts.at(whichPtdata.DIM+whichDim)),2);

```
// 平行後
    #pragma omp parallel for schedule(static) default(none) firstprivate(distance_sum_DIM,DIM
    for (size_t whichPt = 0; whichPt < N_value; whichPt++)
    {
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
            distance_sum_DIM = 0;
            for (size_t whichDim = 0; whichDim < DIM_value ; whichDim++)
            {
                distance_sum_DIM += pow((data.currentCentroids.at(count_k+whichDim*data.K)- c
            }
            // distance_sum_DIM = sqrt(distance_sum_DIM);
            dis[whichPt][count_k] = distance_sum_DIM; //first polo
        }
    }
```

```
    #pragma omp parallel for schedule(static) default(none) firstprivate(N_value,K_value,dis_
    for (size_t whichPt = 0; whichPt < N_value; whichPt++)
    {
        dis_min = FLT_MAX;
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
            if (dis_min > dis[whichPt][count_k] )
            {
                dis_min = dis[whichPt][count_k];
                data.group[whichPt] = count_k;
            }
        }
    }
```

> 跟前面意思一樣其實各個點不需要溝通

UpdateCentroids(迭代區)

- 整個程式的平行核心為 `whichPt` 所以我們要思考的是 在使用 `#pragma omp for schedule(static)` 切割for 迴圈的時候 到底需不需要等
  - 答案是不需要 我們可以把 總量 center_tem 與 總數 counter_tem 分開 後 用 reduction 相加
    `reduction(+:counter_tem[0:data.DIM:K_value])`
    `reduction(+:center_tem[0:data.DIM:K_value])`
  - 每一個dimention 又是獨立的，所以自己的thread做完就不用等了(如果你去等他其中一個 thread if 一直過 那大家不就都要等他，整個speedUP 變成會決定於 loadbalance 問題
  - 所以我們加入 `nowait`

```
// 平行後(AOS)
float   center_tem[DIM_value*K_value] = {0.};
int     counter_tem[DIM_value*K_value] = {0};

#pragma omp parallel default(none) firstprivate(K_value,N_value,DIM_value) shared(data) \
reduction(+:counter_tem[:]) reduction(+:center_tem[:]) proc_bind(spread)
{
    for (size_t whichDim = 0; whichDim < data.DIM; whichDim++)
    {
        for (size_t count_k = 0; count_k < K_value; count_k++)
        {
            #pragma omp for schedule(static) nowait
            for (size_t whichPt = 0; whichPt < N_value; whichPt++)
            {
```

```
                if (data.group[whichPt] == count_k)
                {
                    center_tem[whichDim*K_value + count_k]+=data.pts.at(whichPt*data.DIM+whic
                    counter_tem[whichDim*K_value + count_k]++;
                }
            }
        }
    }
}

// 最後再去跑一個沒有平行的
for (size_t whichDim = 0; whichDim < data.DIM; whichDim++)
{
    for (size_t count_k = 0; count_k < K_value; count_k++)
    {
        data.oldCentroids[count_k+whichDim*data.K]= center_tem[whichDim*K_value + count_k]/fl
    }
}
```

這邊參考老師的

```
// Array section, new int OpenMP 4.5
/// 28b.cpp
#include <iostream>
#include <omp.h>
#include <cstdlib>
using namespace std;
int main(int argc, char **argv)
{
    int votes[5]={0};
    argc>1 ? srand(atoi(argv[1])) : srand(1);    // 反正就是在產生亂數
    auto t1 = omp_get_wtime();
    #pragma omp parallel reduction(+:votes[:])
    {
        for(int i=0; i<1000000; ++i)
        {
            votes[ rand()%5 ] ++;
        }
    }
    t1 = omp_get_wtime() - t1;
    cout << "\nTotal: ";
    for(auto i: votes) cout << i << " ";
    cout << "\nTime spent: " << t1;
    return 0;
}
```

main loop

我們最後輸出

```
    std::cout   << "\n"
                << argv[1] << ", "
                << argv[2] << ", "
                << it << ","
                << timer[0].elapsedTime() << ","
                << timer[1].elapsedTime() << ","
                << timer[2].elapsedTime() << endl;
```

所以我們可以知道

1. K : argv[1]
2. convergence :: argv[2]
3. number of interaction :: it
4. Not include InitializeCentroid :: timer[0]
5. AssignGroups :: timer [1]
6. UpdateCentroids :: timer[2]

繪圖區

每筆資資料為五個取最好(時間最少) 或是平均數值畫圖

- SpeedUP = \frac{Elapsed time of Sequential program}{Elapsed time of presently program}
- Sequential program 表示為平行前的程式

```
aos.exe: src/main.cpp kmeansAOS.o
    $(CC) $(CFLAGS) $< -o $@ kmeansAOS.o

soa.exe: src/main.cpp kmeansSOA.o
    $(CC) $(CFLAGS) $< -o $@ kmeansSOA.o

aos_sequential.exe: src/main.cpp kmeansAOS_sequential.o
    $(CC) $(CFLAGS) $< -o $@ kmeansAOS_sequential.o

soa_sequential.exe: src/main.cpp kmeansSOA_sequential.o
    $(CC) $(CFLAGS) $< -o $@ kmeansSOA_sequential.o
```

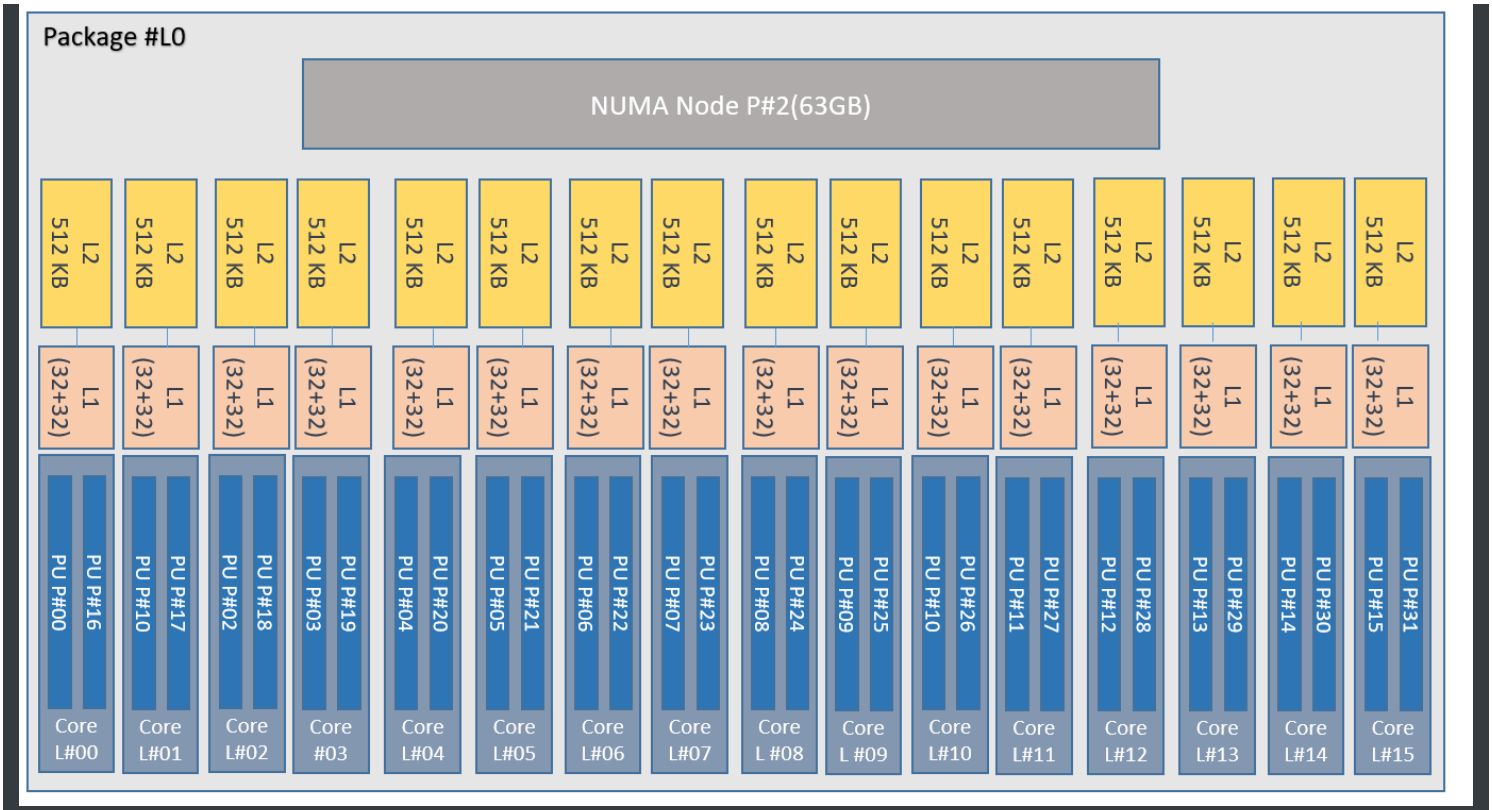
實測

- 為了跟其他人比較這次打算先使用 02 資料來做本次演算法

Information	
資料大小	
noAttributes	3
noLines	245057
使用的K	5
收斂值	1.00E-09
BindingThread	
OMP_PLACES	cores
OMP_PROC_BIND	spread
Firs-touch	ON

- [X]First touch
- [X]export OMP_PLACES=cores
- [X]export OMP_PROC_BIND=spread

> 目前看其他做平行計算的人都是取峰值，所以圖B表示The Best(時間最小)而圖片A則表示平均值，我們可以看到這台機器的電腦架構如下



> 可以看到下表格 Efficienc(SpeedUp / #threads)

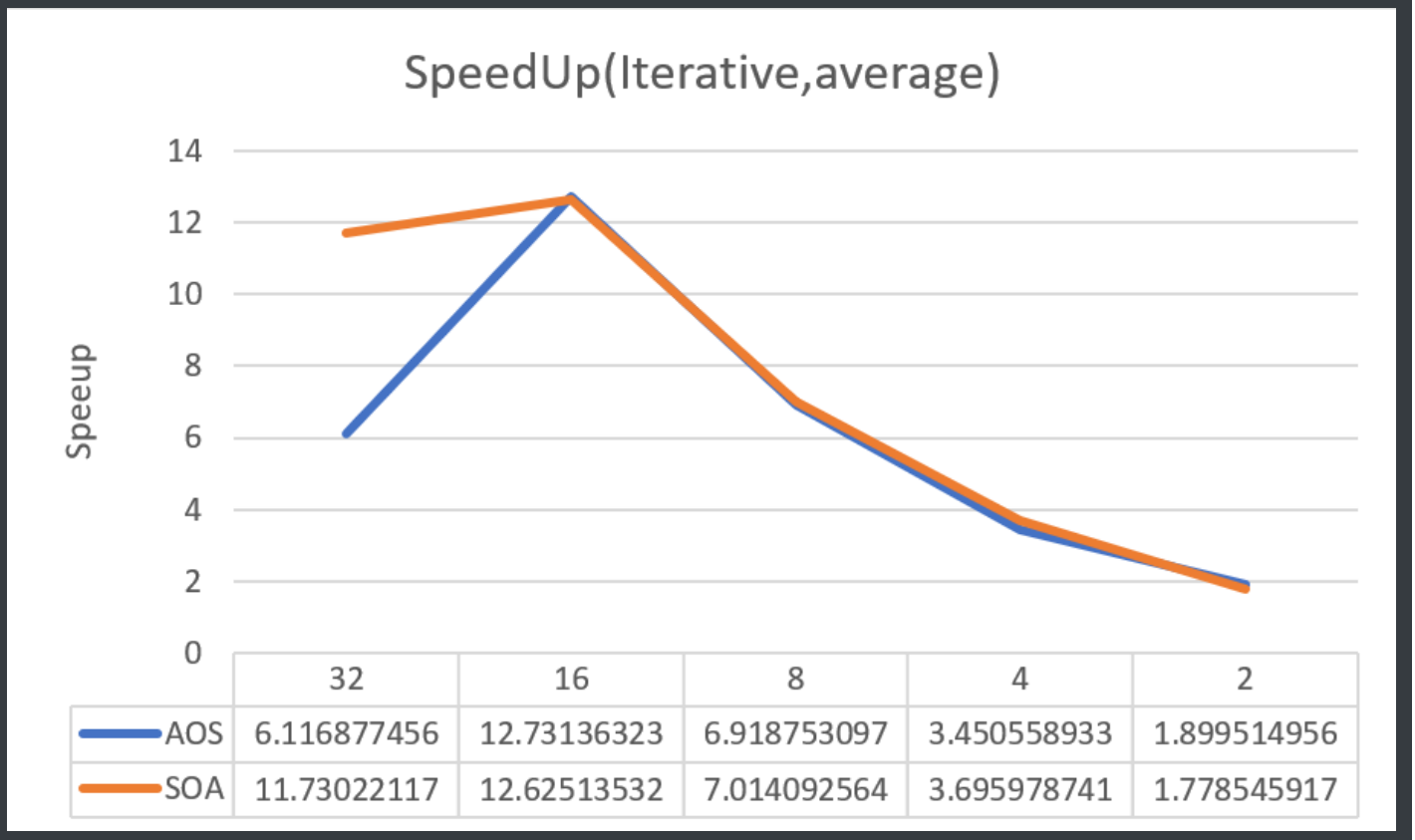
- 參考:圖：B.I

#threads	SpeedUp(Iterative)		Efficienc(Iterative)	
	AOS	SOA	AOS	SOA
2	1.900513777	1.908838825	0.950256888	0.954419413
4	3.668946068	3.710592267	0.917236517	0.927648067
8	6.946244363	7.004289656	0.868280545	0.875536207
16	12.97757978	13.0873584	0.811098736	0.8179599
32	14.58397577	14.53047347	0.455749243	0.454077296

> 我們的 Efficienc 隨著速 thread 超過 16 這台機器的 Core數量效率降低，但是儘管如此我們最在意的SpeedUp還是有從12.9些微提升到 14.58

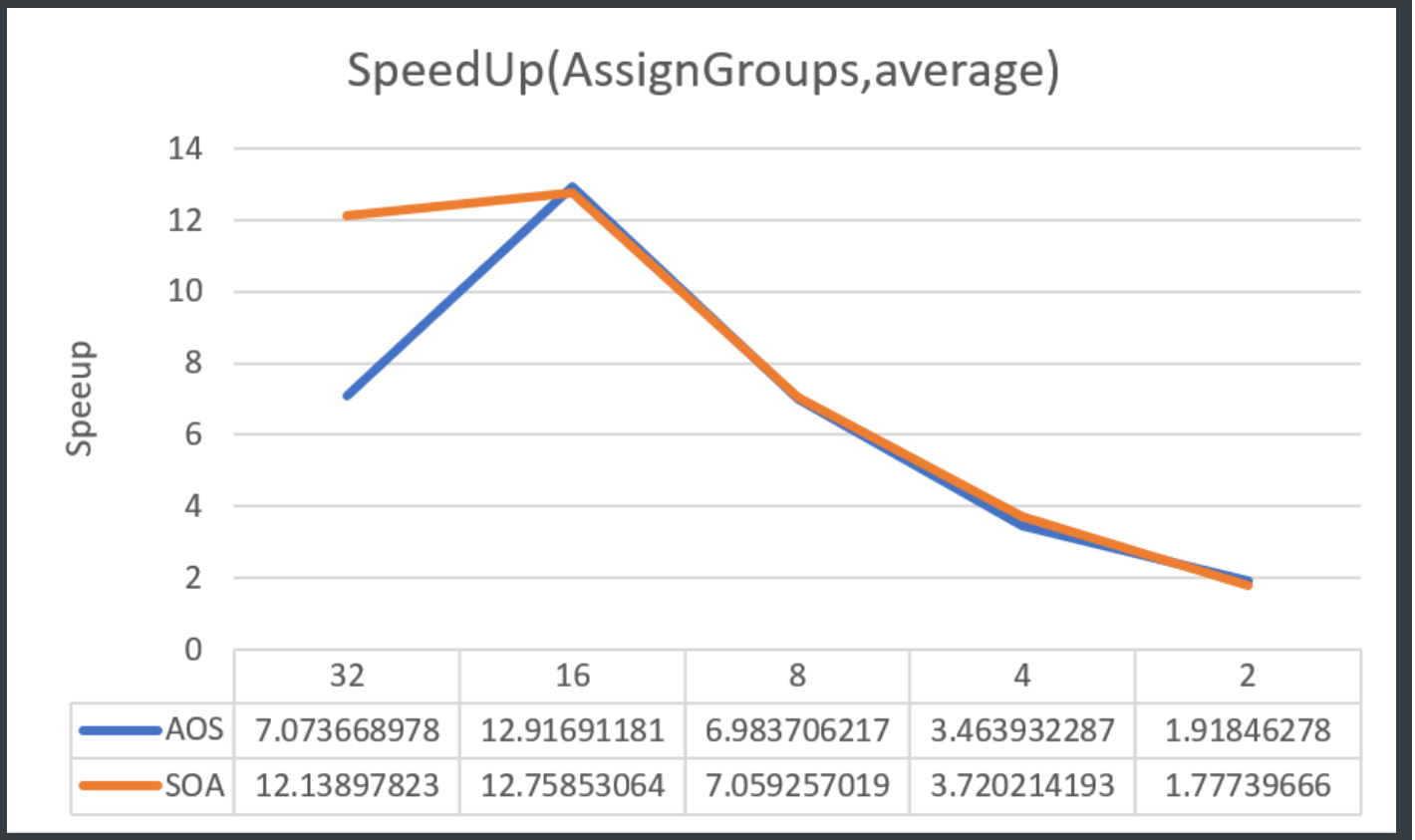
圖A : average

- 圖A.I



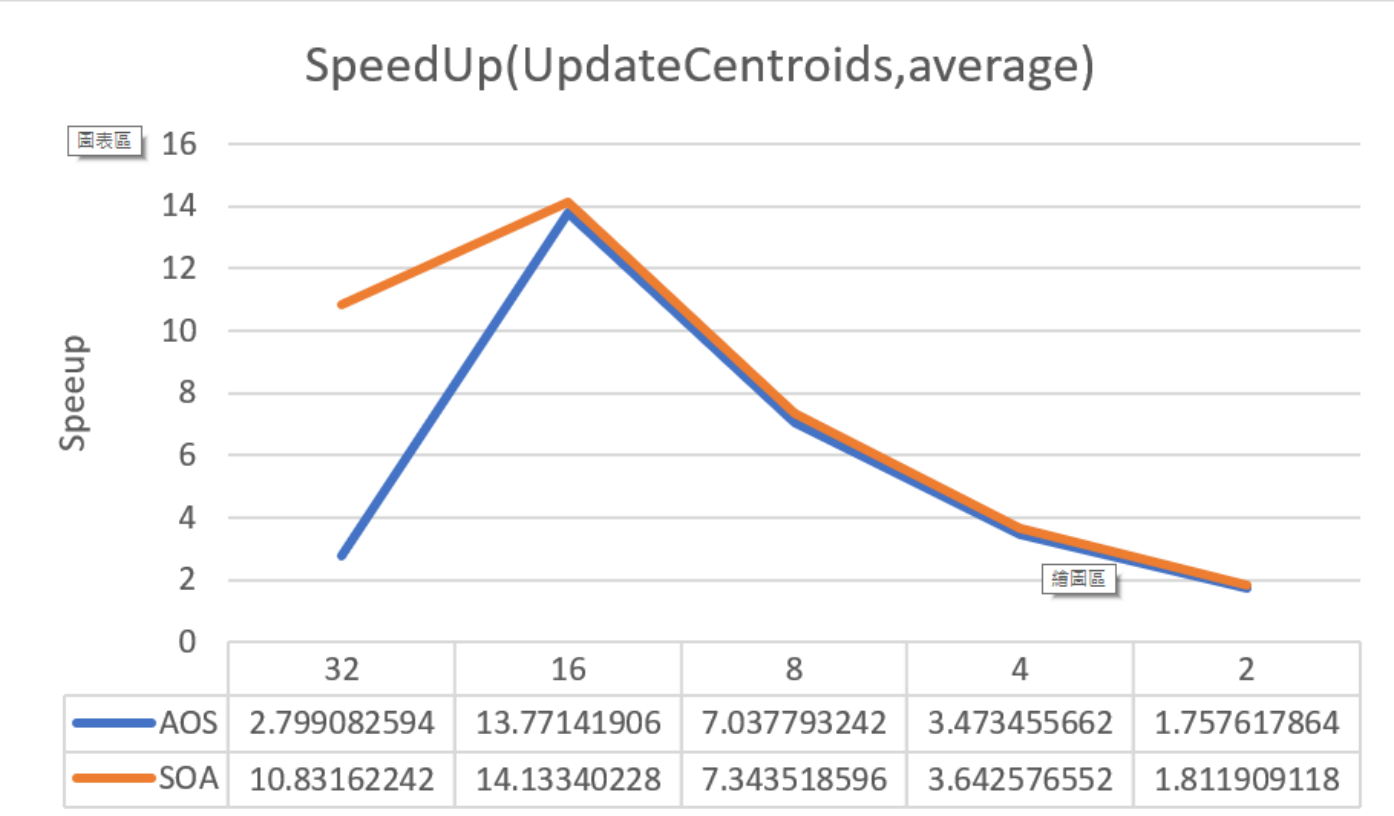
	32	16	8	4	2
AOS	6.116877456	12.73136323	6.918753097	3.450558933	1.899514956
SOA	11.73022117	12.62513532	7.014092564	3.695978741	1.778545917

- 圖A.A



	32	16	8	4	2
AOS	7.073668978	12.91691181	6.983706217	3.463932287	1.91846278
SOA	12.13897823	12.75853064	7.059257019	3.720214193	1.77739666

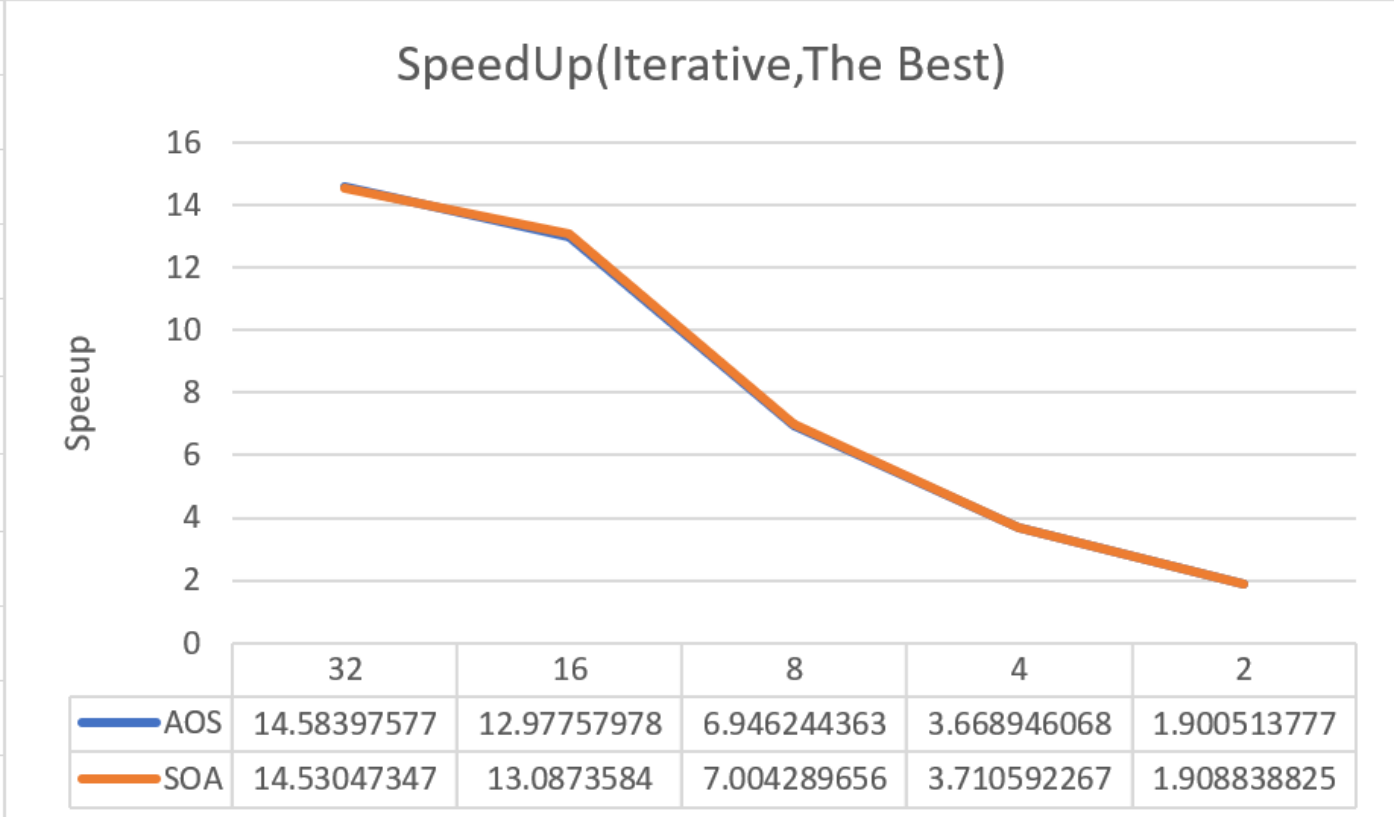
- 圖A.U



	32	16	8	4	2
AOS	2.799082594	13.77141906	7.037793242	3.473455662	1.757617864
SOA	10.83162242	14.13340228	7.343518596	3.642576552	1.811909118

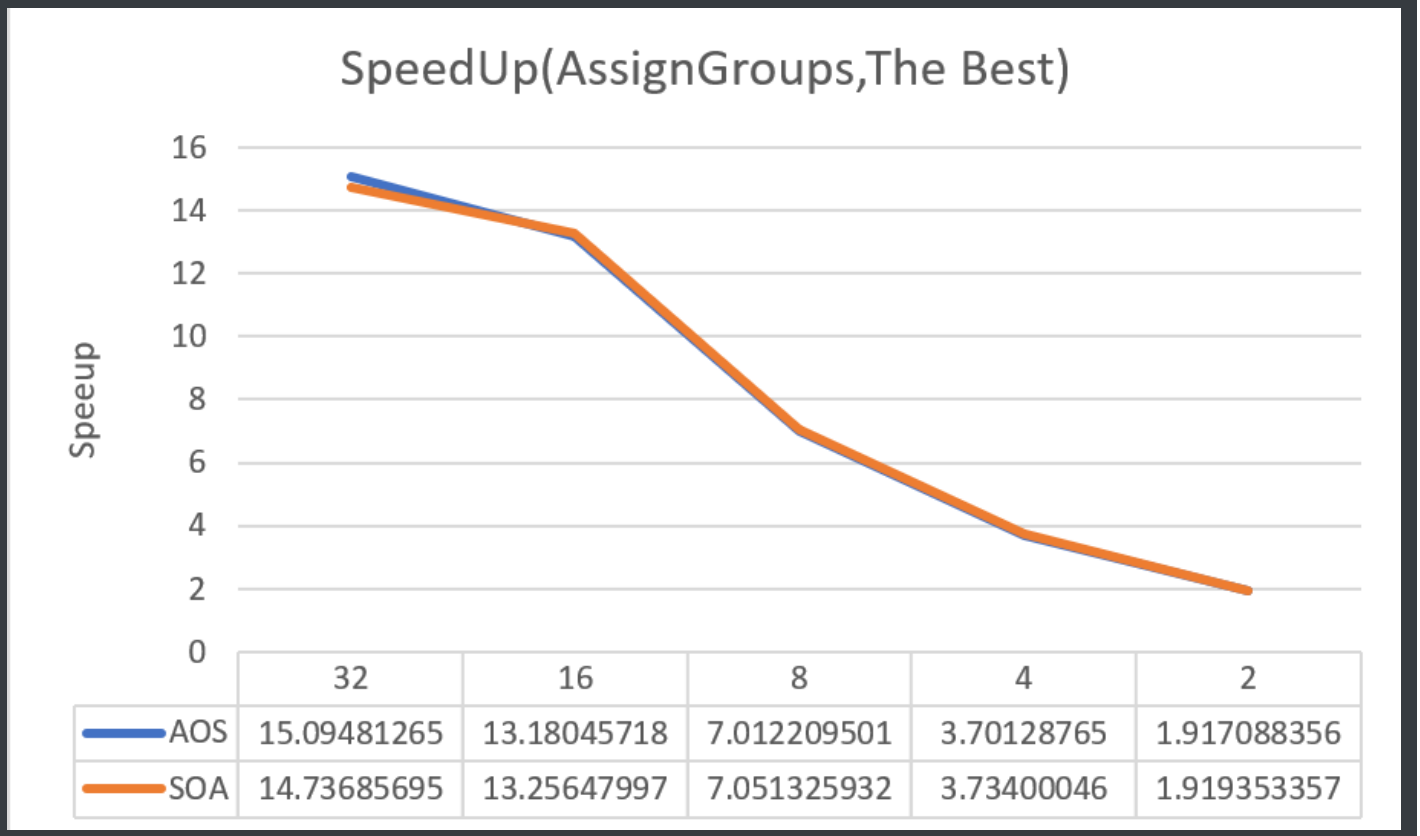
圖B : Best

- 圖B.I



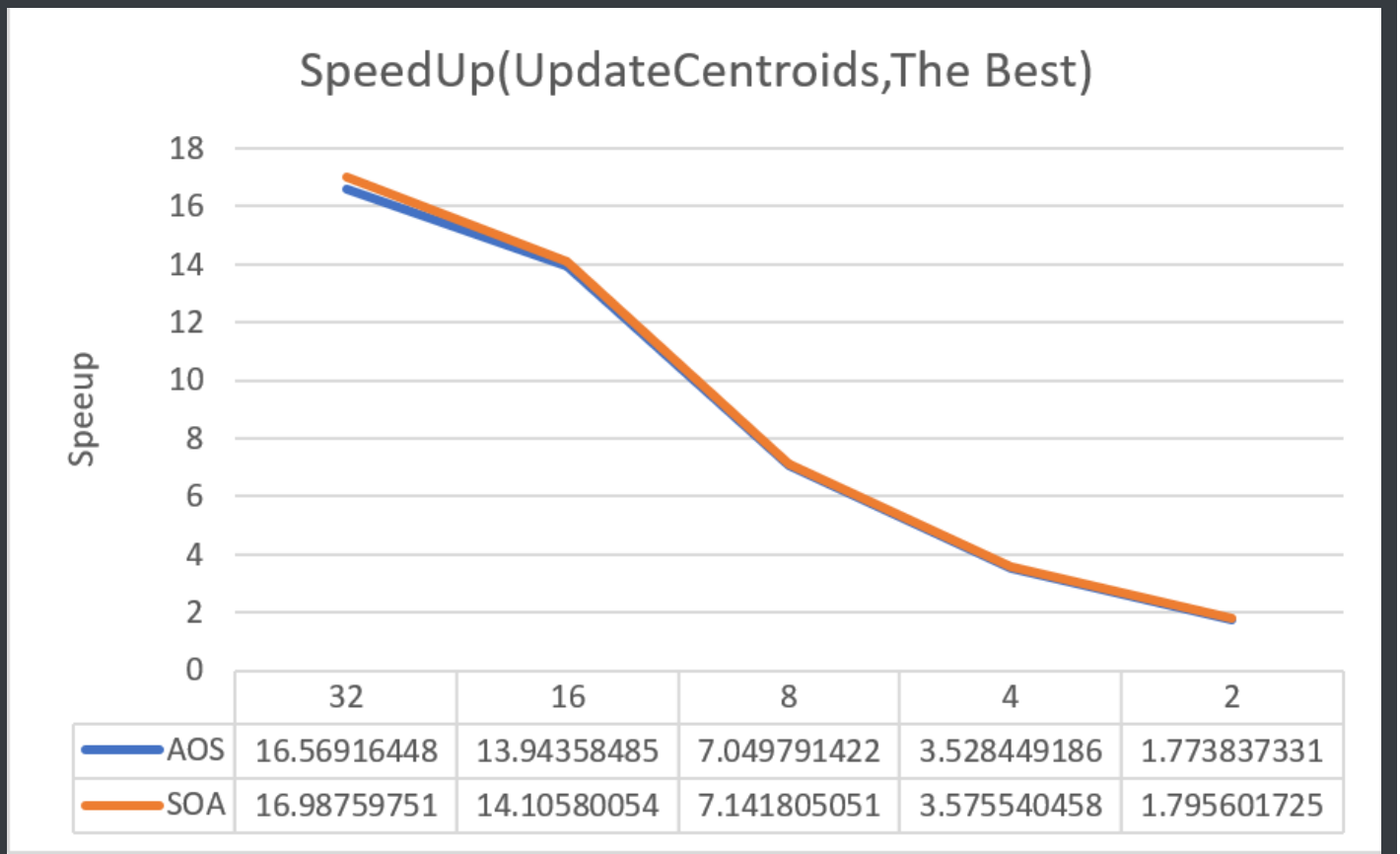
	32	16	8	4	2
AOS	14.58397577	12.97757978	6.946244363	3.668946068	1.900513777
SOA	14.53047347	13.0873584	7.004289656	3.710592267	1.908838825

- 圖B.A



	32	16	8	4	2
AOS	15.09481265	13.18045718	7.012209501	3.70128765	1.917088356
SOA	14.73685695	13.25647997	7.051325932	3.73400046	1.919353357

- 圖B.U



	32	16	8	4	2
AOS	16.56916448	13.94358485	7.049791422	3.528449186	1.773837331
SOA	16.98759751	14.10580054	7.141805051	3.575540458	1.795601725