國立政治大學應用數學系

碩士學位論文

Python 平行化在 SCMDS 上之應用

The application of parallel Python in SCMDS

碩士班學生：李沛承　撰

指導教授：　曾正男　博士

中華民國　102 年　7 月　1 日

Python平行化在SCMDS上之應用

學生： 李沛承　　指導教授： 曾正男博士

國立政治大學應用數學研究所

摘　　要

近年來資料產生的數量遠超過過去可處理的數量，以現今的個人電腦使用傳統的方法已經無法處理大資料的運算與分析，所以改善傳統的方法與平行化為必經的方向，本論文以拆解合成-多元尺度法的平行化為主要討論對象，除了介紹Python程式語言及其相關套件如何撰寫平行化程式，我們將拆解合成-多元尺度法從原本的單核心版本改進為多核心版本，並且探索拆解合成-多元尺度法在平行化過程中的計算效能，藉以了解拆解合成-多元尺度法在平行化計算時的參數要如何設定，使得平行化的SC-MDS可以有最高的計算效率。經實驗證明多核心底下的SC-MDS平行化又把SC-MDS單核心的效能做個再次的提升。

ABSTRACT

In recent years, the number of generated data is growing fast such that it is infeasible to process by using traditional methods. So improving traditional methods and developing paralled computing methods are important issues. The main contribution of this thesis is to delelope the parallel version of the split-and-combine multidimensional scaling method(SC-MDS). We will fistly introduce fundamental python program, the basic python packages and the python multi-core program. Then we will implement the serial core version of SC-MDS to the multi-core version. Moreover, we will discover the efficiency of the multi-core version of SC-MDS. Then we can understand how to determine the parameters of the parllel version of SC-MDS. By our experimental results, we successfully implement the serial core of SC-MDS to the faster parallel version of SC-MDS.

誌　　謝

這次論文能夠在我的預期中順利完成，首先要先感謝我的指導教授曾正男老師，在我剛踏進應用數學系這個領域時，曾老師就常帶領著我一起研究以及不遺餘力地挪出時間指導我的論文，才能夠讓我對數值科學計算的分析那麼快地從無到有，甚至還跟著老師學習平行化的程式碼並且透過老師的指導得以將SC-MDS方法從單核心版本改進成平行化的版本，這段期間有瘋狂、有崩潰不過最讓我得意的是那一份大大的成就感。在此特別感謝口試委員陸行老師以及舒宇宸老師的建議才能使最終版本的論文有更好的結果。

接著讓我體驗到數學的樂趣和奧妙的是陳天進老師，多虧有老師對數學的堅持以及認真的態度，才能讓我比較明白數學的嚴謹以及證明的意義。在這趟數學之旅中還要感謝林澤佑同學兼學長，在許多個實變函數論的夜晚中總是能回答出我要問的一百萬個為什麼。

在讀書以外的時光，要感謝的人實在太多這裡只能意會了，這裡我要感謝研究室的大家，在我最緊張的時候有陪伴我的兄弟江增堂、有讓我學到很多東西的學長李治陞、有常常照顧我和我分享事情的學長詹博翔以及李偉慈、陪伴我打屁吃苦的同學們吳宥柔、高裕哲和陳暐哲以及常常跟我敞開心胸聊天的鄭富元還有一些常常讓我欺負的學弟學妹們以劉宇恩作代表，我知道你(妳)們都是故意逗我開心啦所以我才能認真的跟你(妳)們玩耍，很開心在最後的讀書時光能跟大夥玩得這麼開心，總之真的好喜歡和謝謝各位。最後，我要感謝我家人的包容還有關懷，在我這一大段求學的路上支持和鼓勵我，真是辛苦你們了。

目　　錄

表　目　錄

圖　目　錄

第 一 章　簡介

也許很多人常常聽到平行化這個字眼，但不清楚這到底是做甚麼的、哪種資料才能被平行化亦或是要到哪種平台上運作，對一般人而言要撰寫平行化程式也不是一件容易的事。Python雖然在網路上有許多關於平行化的參考文件，但中文的資料不多並且內容也不容易閱讀，本篇論文將使用Python程式語言為主要工作語言，並提供一個從基礎開始讓大家可以進入平行化計算的參考，文中會提供一些可平行化的範例讓大家由淺入深地了解平行化的真正意義與實際效用。

1.1　為何用Python

如果說為甚麼要用Python這套軟體，不如先來看看它有哪些優點好了 [9]：

直譯式語言 它是一款比較直觀易學的程式軟體，在操作時不用先宣告變數就可以讓變數作運算並且可當做計算機來使用，直譯式也不需要先編譯檔案再執行，讓初學者可以省下許多等待編譯的時間，因此它是一個類似MATLAB和R 的語法。

豐富的標準函式庫和衆多的社群與第三方程式庫 除了Python內建的函式庫已經可以解決大部分的問題外，Python在國外也已發展十幾年了，因此累積了相當多的社群和第三方程式庫，可以說是想得到的函式庫幾乎都有。Ex：如果想作最佳解的問題也可以在網路上搜尋到許多可用的函式庫。

活躍的社群 Python具有非常活躍的社群，不僅有各種討論區還常常有一些活動與演講，並且這個程式語言也不停地在改進中。網路上也有許多Python的社群可以讓初學者在產生問題時提供解決問題的方向以及意見。

物件導向 Python是一款完全物件導向的語言。函式、模組、數字和字串都是物件，並且完全支援繼承、重載、衍生與多繼承，有益於增强原始碼的複用性。因此當程式越大，物件導向的特性也讓Python用起來更得心應手。

跨平台 Python可以當成是跨平台的語言。因為Python直譯的特性，所以任何平台上只要實作直譯器幾乎都可以執行Python，並且目前現有的平台幾乎都有Python的直譯器版本。

被廣泛使用 Python這套軟體在國外早已被廣泛使用，所以其穩定度和受歡迎程度是可想而知的，然而美國太空總署NASA、Google與Youtube ··· 等也都有使用Python，此外還有許多成功的案例，因此Python有機會成為未來更主流的程式語言。

容易擴充和嵌入 Python本身是很好擴充的，如果有非常大量的計算量並且需要速度夠快，這時就可以考慮將負載量大的部分用C語言來寫，然後再用Python來引入就可以加快速度了；然而Python也可以嵌入其它的程式裡面(Ex：MATLAB、OCTAVE)，而這種特性讓Python非常具有彈性。

Python的眾多優點不僅讓使用者更方便外也迅速地將自己推進A級的語言排行榜裡，並且不斷地進步跟爬升名次，爬升的比率從去年到今年就爬升了1.10%，而爬升的比率在這10種編譯語言中僅輸Objective-C。如圖 1.1[4]

Position Mar 2013	Position Mar 2012	Delta in Position	Programming Language	Ratings Mar 2013	Delta Mar 2012	Status
1	1	=	Java	18.156%	+1.05%	A
2	2	=	C	17.141%	+0.05%	A
3	5	⬆⬆	Objective-C	10.230%	+2.49%	A
4	4	=	C++	9.115%	+1.07%	A
5	3	⬇⬇	C#	6.597%	-1.65%	A
6	6	=	PHP	4.809%	-0.75%	A
7	7	=	(Visual) Basic	4.607%	+0.24%	A
8	9	⬆	Python	4.388%	+1.10%	A
9	13	⬆⬆⬆⬆	Ruby	2.150%	+0.74%	A
10	10	=	Perl	1.959%	-0.74%	A

圖 1.1: 編譯語言排行榜(www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

1.2 平行化的需求

到底甚麼是平行化呢，先舉一個例子好了。就像是一對夫妻一起去逛大賣場，進了大賣場後兩個人的各自去買各自的東西，然後結束後一起在出口會合。大家可能會驚訝，不過這就是平行化的基本意義！

那現在我們可以簡單地想像電腦中的平行化，如果要把我們想做的任務丟給電腦處理的話，在過去單核心的電腦中，每個單位時間內一個CPU只能處理一個任務，任務會按照編號依序被處理，但如果是雙核心的電腦，每個單位時間內就能處理兩個任務，速度就會比單核心快上一倍！不過事實上這都是最理想的狀態，因為資料分配給不同的CPU計算需要時間、回收資料也需要時間，所以時間不會是CPU個數的倍數關係，除此之外，也不是所有任務都是可以切割的！很多任務是有關聯性的，這樣如果直接切割給不同的處理核心各自去平行運算，出來的結果肯定是有問題的。多核心的程式在分配、等待、回收的過程中在編寫、維護上，也都比單一核心的程式複雜上不少。那到底什麼樣的程式才是適合平行化的呢，如果說程式中某個迴圈的計算時間很長並且上下迴圈之間可以各自獨立運作，那我們就可以把這個部分分成好多個任務丟給電腦處理，而本篇拿來作平行化的方法為SC-MDS法，因為此方法符合以上的條件，所以拿來處理這個問題是再好不過的了，以下為SC-MDS的基本介紹。

1.3 SCMDS的基本介紹

這一節會簡單地介紹什麼是傳統的多元尺度法(Multidimensional Scaling在此簡稱MDS)[7]以及這個方法的改進也就是拆解合成-多元尺度法(Split-and-Combine MDS在此簡稱SC-MDS)。

多元尺度法(MDS)是一種把高維度資料轉換成低維度資料的方法，在高維度時保持彼此間的距離轉換成低維度時還能維持其資料的結構。這個方法最早是用在如何把地球上城市與城市之間三維的距離位置轉換成二維的資料，由於人類對於高維度的資料難以想像其結構圖形，運用此方法就能讓人們以視覺理解的方式來解讀資料。因此多元尺度法在資料分析上是一個近年來大量被使用的工具。

多元尺度法主要是利用距離矩陣(其中距離矩陣$d[i,j]$代表位置i與位置j之間的距離)經過一些運算以及平移質心後，再透過SVD分解找出我們要的資料，而這些過程的詳細內容會在第3.1節裡介紹。一般來說多元尺度法最主要的過程是SVD，但由於SVD的計算量是$O(N^3)$，所以當矩陣很大時現今的個人電腦大概就無法處理了，所以需要一個可以因應大資料計算的方法，就是拆解-合成多元尺度法(Split-and-combine MDS在此簡稱SC-MDS)。它的核心精神就是把資料拆解成很多塊有重疊的子資料，個別經過多元尺度法後轉換成新的座標位置，再把這些有重疊的新作標位置透過旋轉加以合成(由於重疊的座標位置在原本的資料中是一樣的座標，所以可以旋轉加以合成)，而這些過程的詳細內容也會在第3.1節裡介紹。縱使SC-MDS方法可以加快MDS的計算速度，然而SC-MDS的運算邏輯上有許多很適合平行化的部分，因此本論文希望藉由Python程式很容易撰寫平行化程式的特性來改良現有的SC-MDS方法。

本論文的架構為第一章為基本介紹、第二章為介紹Python之平行計算，主要是介紹Python關於平行化的一些函式與其使用方法的簡介、第三章為介紹傳統的多元尺度法(MDS)會詳細地介紹其推導流程以及介紹為何使用拆解合成-多元尺度法及其推導流程，並附上單核心版本以及平行化版本的程式碼，第四章為實驗結果討論拆解合成-多元尺度法及其平行化在多核心底下的執行效率以及第五章為本論文的結論。

第 二 章　Python之平行計算

2.1　Python基本運算之套件與工具

在網路上有許多Python入門的電子書[12][14][18]、相關知識[16]以及書籍[1][2]在這裡提供給讀者做參考，由於篇幅的關係所以詳細的使用方法以及介紹就不在這裡多作說明了。這一章節介紹一些Python內常用的套件，有了這些套件後大部分的問題幾乎都可以處理了；而且使用這些套件來進行科學計算，還能獲得世界各地的開發者提供的套件資源。以下就是上述所提到的套件 [8]：

Numpy 是一種Python語言的延伸，其中包含了定義數值陣列和矩陣類型與一些在上面的基本操作。這裡會介紹到兩個例子，主要的目的是要表現出不用Numpy這個套件比起用了Numpy這個套件在處理一些簡單的基本操作中哪個速度會比較快，例子2.1-1會介紹到Python中不用Numpy這個套件的例子、例子2.1-2則會介紹到Python中用了Numpy這個套件的例子。

例子2.1-1：這是不用Numpy裡陣列的方式使得數值相加。

```
# add.py
# 把以下為add.py的程式碼
01| import time
  # 引入time套件
02| currtime = time.time()
  # 起始時間。把此刻的執行時間記錄下來儲存到currtime中
03| a = range(10000000)
  # a是一個串列，其中第1個位置是0、第2個位置是1、以此類推到第10000000個位置是9999999，因此串列的大小是10000000
04| b = range(10000000)
  # b與上述的a一樣
05| c = []
  # c是一個空的串列，等等要把a與b的值相加並放入其中
  # 由於兩個串列並不能直接相加，所以要透過for迴圈把個別位置的值相
```

加後並放入c這個空的串列中

```
06| for i in range(len(a)):
  # for迴圈跑的次數是a這個串列的大小，而i的值會根據迴圈的次數依序從0跑到9999999
07|     c.append(a[i] + b[i])
      # 每一次的迴圈中會把a[i]與b[i]的值作加總並放入c這個串列中
08| print time.time() - currtime
  # 把此刻的執行時間減去剛剛的起始時間則為整段程式的運行時間，並把運行時間顯示出來
```

```
# 以上為add.py的程式碼

09| >>> import add
  # 執行add
    5.51968598366
  # 運行時間為5.51968598366(sec)
```

例子2.1-2：這是用Numpy裡陣列的方法讓數值相加。

```
# add1.py
# 把以下為add1.py的程式碼
01| import time
  # 引入time套件
02| import numpy as np
  # 把numpy更名為np
03| currtime = time.time()
  # 起始時間。把此刻的執行時間記錄下來儲存到currtime中
04| a = np.arange(10000000)
  # a是一個np中的陣列，其中第1個位置是0、第2個位置是1、以此類推到第10000000個位置是9999999，因此陣列的大小是10000000
05| b = np.arange(10000000)
  # b與上述的a一樣
```

```
06| c = a + b
  # 由於np中的陣列是可以直接相加的，所以c為陣列a與陣列b的加總
07| print time.time() - currtime
  # 把此刻的執行時間減去剛剛的起始時間則為整段程式的運行時間，並把運行時間顯示出來
-----------------------------------------------------------------
# 以上為add1.py的程式碼

08| >>> import add1
  # 執行add1
    0.0789999961853
  # 運行時間為0.0789999961853(sec)
```

其結果可發現例子2.1-2所花的時間比例子2.1-1少上許多，時間從5.51968598366(sec)→0.0789999961853(sec)，原因在於用for迴圈相加時程式會去判斷每次運算時的型態而導致速度有所延誤，但是用上Numpy裡的陣列相加時其結果幾乎是立即運算出來的，而且Numpy這個套件不僅能簡化程式碼還可以減少運算時間，這就是Numpy的好用之處。而除了這個功能外，Numpy裡還有許多其它的功能也是很好用的，有興趣的讀者可以直接瀏覽官網來學習。[8]

SciPy 是另一種Python語言的延伸，其中使用Numpy作高階的數學運算、訊號處理、優化、統計數據等等。接下來例子2.1-3會介紹三種引用套件的方法，並介紹這些方法的優劣。

例子2.1-3：

(方法一)

```
01| >>> from scipy import *
  # 從scipy這個套件中把所有的工具都引入了
02| >>> a = zeros(1000)
  # 這時就可以直接使用scipy套件中的zeros這個功能，並且a為一條1000維的零向量
  # 把整個套件都引入進來，所以也會引入很多不太用的套件。
```

(方法二)

```
01| >>> from scipy import abs, sin, pi, dot, asarray, diag, zeros, empty
  # 從scipy這個套件中只引入了abs, sin, pi, dot, asarray, diag, zeros, empty
這些工具
02| >>> a = zeros(1000)
  # 這時就可以直接使用scipy套件中的zeros這個功能，並且a為一條1000維
的零向量
  # 雖然只引入用到的套件，但是有點太過冗長。
```

(方法三)

```
01| >>> import scipy as sp
  # 引入scipy整個套件並把scipy更名為sp
02| >>> a = sp.zeros(1000)
  # 這時要使用sp這個套件中zeros這個功能時就要寫成sp.zeros，並
且a為一條1000維的零向量
  # 這也是本文比較常用的方法，並且在Python寫作查詢方面就不會有一
大堆的函式，比較方便使用。
```

Matplotlib 這個套件是模仿MATLAB裡的繪圖套件，是一種Python語言的延伸來方便繪圖。以下的例子則是使用這個繪圖套件畫出sin函數圖形的範例。

例子2.1-4：

```
01| >>> import scipy as sp
  # 引入scipy套件並更名為sp
02| >>> from matplotlib.pylab import *
  # 從matplotlib.pylab裡引入所有工具
03| >>> a = sp.arange(0,2*pi,0.01)
  # a是一個一維陣列，裡面的值是從0到2*pi以0.01為間距的數
04| >>> b = sp.sin(a)
  # b是一個一維陣列，裡面的值是把a的值帶入sin函數後的結果
05| >>> plot(a,b)
  # 建立一個a與b對應位置的圖
```

```
06| >>> show()
  # 把圖展示出來
```

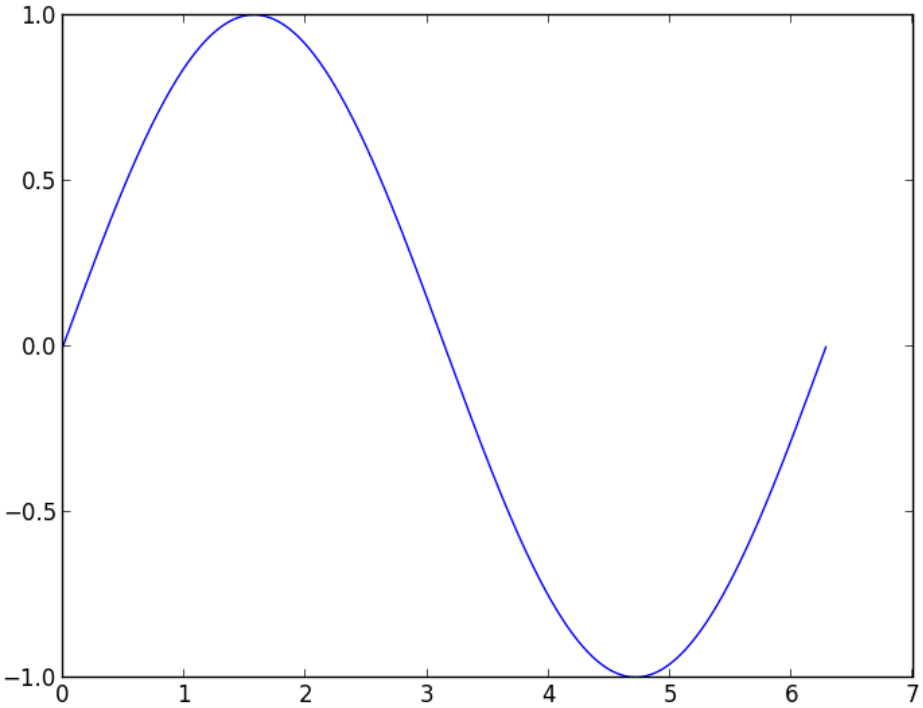


圖 2.1: 例子2.1-4 sin函數

2.2 Python平行計算之套件與相關介紹

在介紹了基本的科學計算套件之後，這一節會介紹到平行化的套件，以及帶入單核心轉平行化的例子，讓讀者更好理解如何平行化。

multiprocessing 這是一個提供平行化功能的套件，可以有效地控制process達到平行運算的目的。其底下有幾個常用到的工具先在這邊做一個簡單的介紹，介紹完後會帶入有程式碼的範例讓讀者能更好理解。其工具如下 [11]：

1.cpu_count 回傳系統中CPUs的數目。

2.Pool 回傳一個process pool的物件。

3.Process 用此物件來指示要用幾個分別的process執行當前的程式。

4.Array 製造一個同步的共享陣列。

5.Lock 讓平行進行中的process等待其所需要同步的結果。

例子2.2-1：我們先作個比較基礎的範例，這個例子要把以下這個函數轉成平行化來計算，方法一是原本單核心的方式；方法二是使用multiprocessing的套件改成平行化(多個process)的方式，並比較在不同的process個數底下速度的差別。

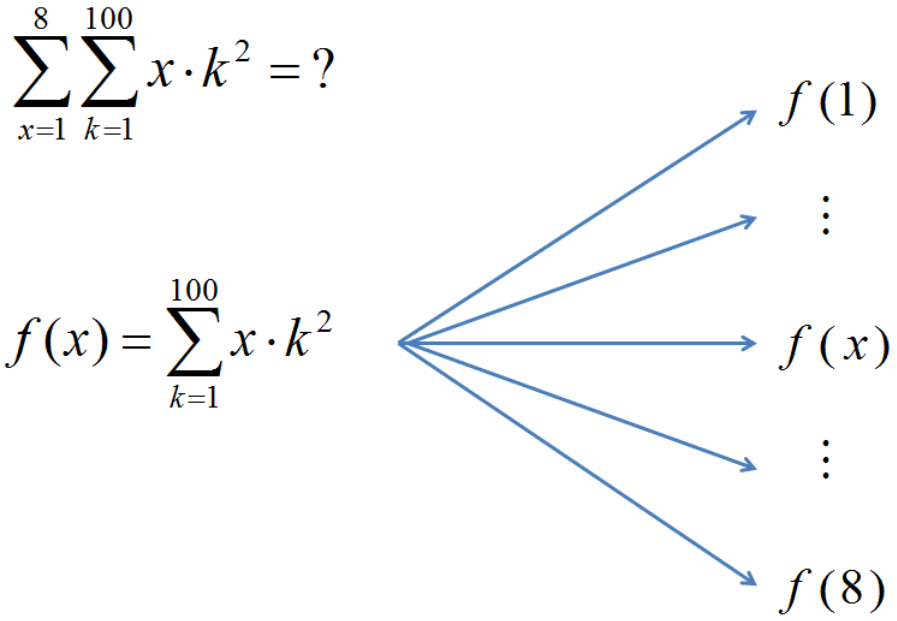


圖 2.2: 例子2.2-1

(方法一：Python serial code)

```
01| K = 100
  # 宣告接下來的迴圈中k的上界為100
02| N = 8
  # 宣告接下來的迴圈中x的上界為8
03| w = 0
  # 宣告w是第一個迴圈中結果的加總，起始值是0
```

```
for x in range(1,N+1):
    r = 0
    # 宣告r是第二個迴圈中結果的加總，起始值是0
    for k in range(1,K+1):
        r += x*(k**2)
        # 在這個迴圈中r加上了x的k平方次，其中x為1~8中某一個數字，並且k為1到100
    w += r
    # 在這個迴圈中，w把上一行所計算的r值加總
    # 這裡把r與w分成兩段寫的用意是讓讀者可以比較直觀的把單核心版本改成平行化版本
# 此為非平行的版本
```

(方法二：Parallel python code)

```
# simple.py
from multiprocessing import Pool
# 從multiprocessing這個套件中引入Pool這個工具
import time
# 引入time這個套件
K = 100
# 宣告接下來的迴圈中k的上界為100
def Fx((x,)):
# 這個Fx函式是要拿來計算平行化部分時的值，也就是方法一中第二個迴圈中作的事，如圖2.2
    r = 0
    # 宣告r是Fx函式中結果的加總，起始值是0
    for k in xrange(1, K+1):
        time.sleep(0.01)
        # 由於這個函式的計算量非常小，所以在這個迴圈中的每次停滯0.01秒是為了讓時間拉長，並比較在不同個數的process底下到底有沒有平行化
```

```
08|         r += x*(k**2)
          # 在這個迴圈中r加上了x的k平方次，其中x為1~8中某一個數字，並且k為1到100
09|     return r
      # 回傳r值
10| def sum_Fx(N = 8,ncpu = 4):
  # 這個sum_Fx函式是要把N個平行化Fx計算出來的值作一個加總，其中N的預設值(x的上界)是8，ncpu的預設值(process的個數)是4
11|     if __name__ == "simple":
      # 如果這個檔案的檔名是simple則作以下的事情
12|         currtime = time.time()
          # 起始時間。把此刻的執行時間記錄下來儲存到currtime中
13|         po = Pool(processes = ncpu)
          # 開process的數量為ncpu個，輸入端設預設值為4個
14|         res = po.map_async(Fx,((i,) for i in xrange(1,N+1)))
          # 用po.map_async這個工具可以把Fx平行計算接著把值回傳出來儲存在res裡
15|         w = sum(res.get())
          # 用sum把res的值加總在w裡
16|         print time.time() - currtime
          # 把此刻的執行時間減去剛剛的起始時間則為整段程式的運行時間，並把運行時間顯示出來
17|         return w
          # 把結果w回傳出來
-----------------------------------------------------------------
# 以上為simple.py的程式碼，此為平行化的版本

18| >>> import simple
  # 引入simple這個py檔
19| >>> simple.sum_Fx(8,1)
  # 執行simple.sum_Fx(8,1)
```

```
    8.11127901077
  # 運行時間為8.11127901077(sec)
    12180600
  # 這個函數的結果為12180600
20| >>> simple.sum_Fx(8,2)
  # 執行simple.sum_Fx(8,2)
    4.06410813332
  # 運行時間為4.06410813332(sec)
    12180600
  # 這個函數的結果為12180600
21| >>> simple.sum_Fx()
  # 執行simple.sum_Fx()，由於預設值N是8、ncpu是4等同執行simple.sum_Fx(8,4)
    2.04466485977
  # 運行時間為2.04466485977(sec)
    12180600
  # 這個函數的結果為12180600
```

simple.sun_Fx(”第一個值”,”第二個值”)中”第一個值”代表的是x的範圍，在這裡就是1～8；”第二個值”代表的是要分成幾個process來跑。所以simple.sum_Fx(8,1)則表示x的範圍從1～8且只用一個process來跑，換句話說simple.sum_Fx(8,2)就是用兩個process來跑，而時間從 8.09999990463(sec)→4.08899998665(sec)，以此類推。

在此版本中的time.sleep(0.01)會讓執行此副函式『Fx((x,))』的process每執行一次則有個停滯時間以代替大計算量時的執行時間，而要檢查是否有分工給多個process同時運作時，當分工的process變多時運行時間應該降低，這一點可從上述的例子則會得到驗證，當執行simple.sum_Fx(8,1)(一個process)、simple.sum_Fx(8,2)(兩個process)與simple.sum_Fx()(4個process)的運行時間就從8.11127901077(sec)→4.06410813332(sec)→2.04466485977(sec)，當process變多時運行時間真的有降低，可見multiprocessing這個套件真的有平行化的功能。

接著是本篇的重點之一，在平行計算大資料時常常需要把某段資料丟入個別要計算的副函式當中，再經由各自取值把計算後的資料重組，這不僅僅浪費過

多記憶體來儲存值還會增加組合時的複雜性。所以本篇介紹一個共享記憶體的方法**shared_array**：這是開創一個共享的陣列以方便平行運算時資料的計算和存放，簡單的說可以當作一個global域的共享陣列來使用。以下則為shared_array的宣告方式：

shared_array

```
01| import multiprocessing as mp
  # 引入multiprocessing套件並更名為mp
02| import numpy as np
  # 引入numpy套件並更名為np
03| import ctypes
  # 引入ctypes套件
04| class shared_array(object):
  # class是一種『類別』的寫法，它底下(程式碼05~10行)則會定義出這個類別該有的屬性
05|   def __init__(self,matrix_shape=(1,1)):
    # 這裡把shared_array這個共享陣列的預設大小為1*1的陣列
06|     (m,n) = matrix_shape
      # 如果輸入端有放入值(m,n)時，這個陣列的大小會是輸入端的m*n
07|     self.shared_base = mp.Array(ctypes.c_double,m*n)
08|     self.array = np.ctypeslib.as_array(self.shared_base.get_obj())
      # 07、08行是開一個共享的零陣列(裡面所有的元素都是零)，陣列大小則是輸入端的m*n
09|     self.array = self.array.reshape(m,n)
      # 如果宣告完成後，『新的變數名』.array則會顯示這個shared\_array的陣列中m*n的值
10|     self.shape = self.array.shape
      # 如果宣告完成後，『新的變數名』.shape則會顯示這個shared\_array的陣列大小
# 程式碼的01~10行完成了shared_array這個class的宣告，之後就可以使用這個class開啓共享的零陣列
# 範例如下
```

```
11| >>> A = shared_array()
  # 讓A成為一個共享的零陣列，shared_array的輸入端不填值則會是一個1*1的零陣列
12| >>> A.array
  # 展示A的陣列內容
    array([[ 0.]])
  # 1*1的零陣列
13| >>> A = shared_array((2,4))
  # 讓A成為一個2*4的共享零陣列，
14| >>> A.array
  # 展示A的陣列內容
    array([[ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.]])
  # 2*4的零陣列
```

在這裡的宣告共享陣列**shared_array**只用零陣列是為了方便先預開一個共享的陣列大小再把值依序填入，用法跟numpy裡的zeros很像不過是共享的。接續上述的程式碼，把共享的零陣列開好後本篇也提供了一個的副函式**array2shared**讓值可以方便地複製到共享陣列中。

array2shared

```
15| def array2shared(A,shared_A):
  # array2shared這個副函式的輸入端是A與shared_A，它的功能是把A陣列的值複製到shared_A這個共享陣列中
16|   m1,n1 = A.shape
    # m1,n1為A陣列的大小
17|   m2,n2 = shared_A.shape
    # m2,n2為shared_A共享陣列的大小
18|   if m1<>m2 or n1<>n2:
    # 這個if判斷式是要保證m1=m2以及n1=n2，如果大小不一樣時則執行以下敘述
```

```
19|     print 'The size of matrices must be the same'
20|     return
21|   else:
    # 如果m1=m2以及n1=n2時，則執行以下敘述
22|     for i in range(m1):
23|       shared_A.array[i] = A[i]
        # 把A的第i列複製到shaed_A.array的第i列
# 範例如下：

24| >>> A = np.random.random((2,4))
  # A為隨機產生的2*4陣列，其中值從0~1的浮點數中隨機中挑出來
25| >>> A
  # 展示A
    array([[ 0.6659476 ,  0.84162458,  0.56063487,  0.13024106],
           [ 0.77196049,  0.84924812,  0.32920407,  0.50888212]])
  # A為隨機的2*4陣列
26| >>> shared_A = shared_array((2,4))
  # shared_A為2*4的共享零陣列
27| >>> shared_A.array
  # 展示shared_A.array
    array([[ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.]])
  # shared_A為2*4的共享零陣列
28| >>> array2shared(A,shared_A)
  # 把A陣列的值複製到shared_A共享陣列中
29| >>> shared_A.array
  # 展示shared_A.array
    array([[ 0.6659476 ,  0.84162458,  0.56063487,  0.13024106],
           [ 0.77196049,  0.84924812,  0.32920407,  0.50888212]])
  # A的值已完全的複製到shared_A裡了
```

接續上述的程式碼，既然共享的陣列已經可以方便的開創和使用，本篇要作的平行計算也就能開始動工了，在這之前先介紹multiprocessing套件裡一個特殊的功能**Lock**，使用這個工具就能讓多個process在平行計算中等待別個process所需要同步與執行的結果，再接著繼續完成它的任務。

Lock 例子2.2-2：這個例子主要是要表達出當副函式裡大計算量的地方時可以讓多個process同時平行處理，而需要同步與小計算量的地方時再使用**Lock**的功能讓process作一個等待與同步的動作。而這個副函式需要同步的地方則是當副函式被呼叫時的次數(count.array)也會加入計算，這個例子會使用到上述shared_array、array2shared與Lock的應用。如圖 2.3：

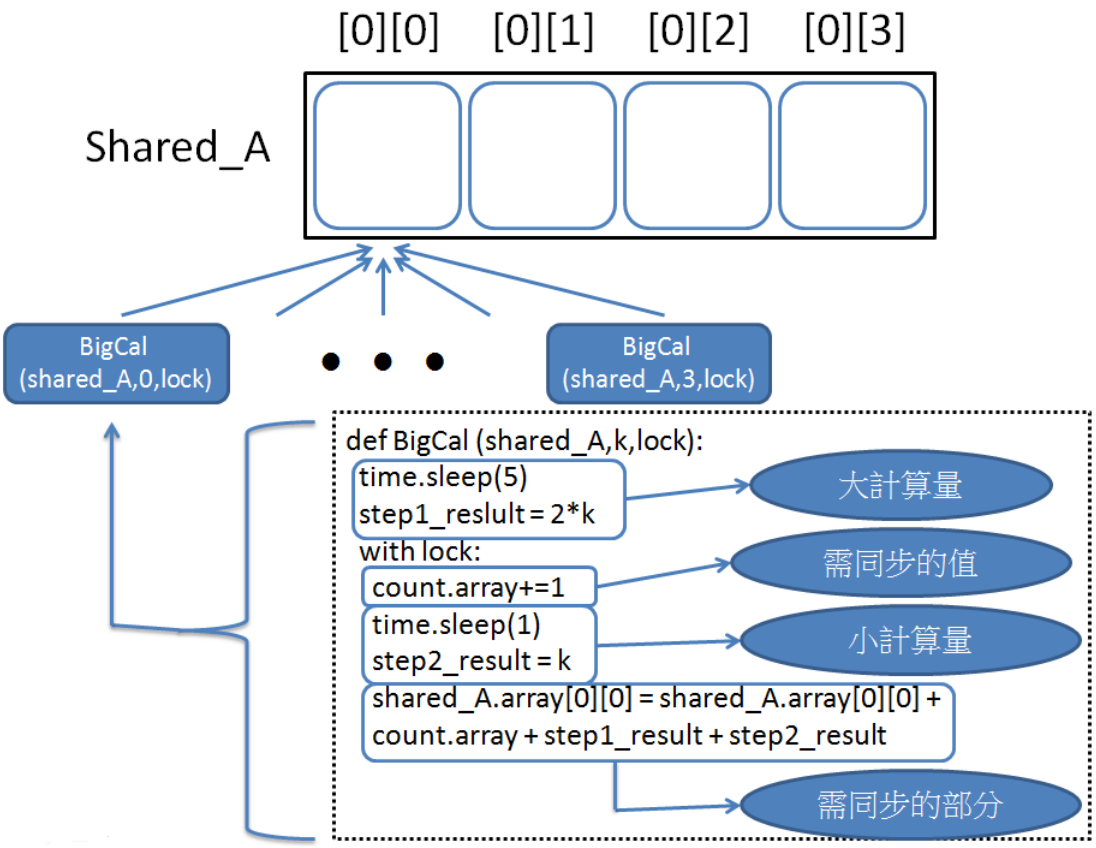


圖 2.3: 例子2.2-2 BigCal

```
import time
# 引入time套件
def Fun_A(A,k):
# Fun_A這個副函式中會把k個index個別作運算，再回傳出來
  start = time.time()
  # 起始時間。把此刻的執行時間記錄下來儲存到start中
  def BigCal(shared_A,k,lock):
  # BigCal這個副函式等等會被平行運算，個別運算後並把值存到shared_A裡
    time.sleep(5)
    # 這裡的time.sleep(5)是讓時間時間停滯5秒以此模擬大計算量時的執行時間
    step1_reslult = 2*k
    # 而step1_result則是模擬大計算量完成的結果
    with lock:
    # 這裡的with lock功能會讓第一個執行到此行的process繼續執行以下的程式碼，直到此process執行結束才會讓別的proces執行
      count.array+=1
      # 這裡的count當作計數器記錄這個副函式『BigCal』執行了幾次
      time.sleep(1)
      # 這裡的time.sleep(1)是讓時間時間停滯1秒以此模擬小計算量時的執行時間
      step2_result = k
      # 而step2_result則是模擬小計算量完成的結果
      shared_A.array[0][0] = shared_A.array[0][0] + count.array + step1_result + step2_result
    # 這裡的shared_A.array[0][0]是需要被同步的部分，其中step1_result是可以同時間獨立給每個process執行的部分，而count.array則是需要等待被同步的值
  m,n = A.shape
  # m與n是A陣列的大小
```

```
42|   shared_A = shared_array((m,n))
    # shared_A是一個m*n的共享零陣列
43|   assert shared_A.array.base.base is shared_A.shared_base.get_obj()
    # 讓shared_A在執行結束後把共享記憶體釋放掉，不然會一直佔著記
憶體的空間
44|   array2shared(A,shared_A)
    # 把A陣列的值複製到shared_A共享陣列中
45|   count = shared_array((1,1))
    # count是一個1*1的共享零陣列
46|   assert count.array.base.base is count.shared_base.get_obj()
    # 讓count在執行結束後把共享記憶體釋放掉，不然會一直佔著記憶體
的空間
47|   lock = mp.Lock()
    # 令lock為mp裡的Lock功能，也就是會等待前一個process完成工作的
功能
48|   procs = [mp.Process(target=BigCal, args=(shared_A,i,lock)) for
      i in range(k)]
    # 這裡把要平行運算的副函式放入procs這個list裡
49|   for p in procs: p.start()
50|   for p in procs: p.join()
    # 此為執行procs整個list
51|   return shared_A.array,time.time()-start
    # 回傳執行結果與運行時間
  # 範例如下：

53| >>> A = np.zeros((1,4))
  # 令A為1*4的陣列
54| >>> A
  # 展示A
    array([[ 0.,  0.,  0.,  0.]])
  # 1*4的零陣列
```

```
55| >>> shared_A,time = Fun_A(A)
  # 令shared_A、time為Fun_A(A)的兩個輸出值
56| >>> shared_A[0][0]
  # 展示shared_A[0][0]
    28.0
  # 執行結果為28.0
57| >>> time
  # 展示time
    9.020509004592896
  # 運行時間為9.020509004592896(sec)
```

由於process在執行副函式『BigCal』裡的time.sleep(5)會讓時間停滯5秒(在此為模擬大計算量的時間)，接著副函式『BigCal』裡的with lock功能會等待共享記憶體上其他還沒執行結束的任務，在這裡分成了4個process每個都會time.sleep(1)所以執行時間是9.020509004592896(sec)，shared_A[0][0]的結果則會是28.0。而如果把with lock的功能拿掉，4個process則會同時執行『BigCal』所以執行時間是6.01941990852356(sec)，shared_A[0][0]的結果會是34.0，時間從9.020509004592896(sec)→6.01941990852356(sec)，結果也從28.0→34.0，原因是有with lock功能時count被執行有先後順序所以在4個process中的count值是1、2、3、4，而沒有with lock功能時4個process會同時執行count+=1使得count值都是4。可發現with lock的確具有讓其他process等代的功能所以從上述例子中可以得到證明。例子2.2-2已經可以操作共享記憶體並且還能使平行運算達到等待的功能，接下來要介紹的是該如何控制電腦的核心，並且觀察在不同核心數底下的執行效能。

2.3 Python控制核心之套件

除了平行運算之外，多核心操作的效能也是本篇所要探討的重點之一，在這一節會介紹控制核心的套件以及如何操作它。

affinity 這是一個提供控制核心的套件，可以控制核心使用的數目[10]。

例子2.3-1：介紹如何知道電腦當下核心的使用數目與控制核心的數目。

```
01| >>> import affinity
  # 引入affinity套件
02| >>> affinity.get_process_affinity_mask(0)
  # 查看當下核心的使用數目
    65535
  # 此輸出值為2的『核心數目』次方再減1，所以65535這個值為2的『16』次方再減1，故當下核心數目為『16』個
03| >>> affinity.set_process_affinity_mask(0,2**8-1)
  # 此輸入端的第二項為2的『核心數目』次方再減1，所以2**『8』-1則代表現在控制核心的使用數目為『8』個
```

在此令affinity.get_process_affinity_mask(0)的輸出值為K，由以上的例子我們可以得到這個公式，$K = 2^n - 1$，其中K代表的是輸出值、n代表的是『核心數目』；換句話說如果我們需要控制核心數目的時候，我們就可以使用affinity.set_process_affinity_mask(0,2**n-1)，其中n代表的就是『核心數目』。有了這個套件之後，我們就可以方便地觀察在不同的核心數底下，多個process的平行化在執行效能上到底有沒有影響，而執行效能的比較在第四章則會作比較詳細的介紹。基本的操作在這一個章節告一段落，在之後的篇幅會介紹到平行計算在其他地方上的應用。

第 三 章　SCMDS與其平行化

在這一章節會介紹到傳統的多元尺度法(Multidimensional Scaling在此簡稱MDS)與拆解合成-多元尺度法(Split-and-Combine MDS在此簡稱SC-MDS)的概念以及哪裡有需要平行化的地方、如何放入平行化的方法加快其速度。

3.1　SCMDS以及單核心版本 for Python

多元尺度法(MDS)是一種把高維度資料轉換成低維度資料的方法，在高維度時保持彼此間的距離轉換成低維度時還能維持其資料的結構。這個方法最早是用在如何把地球上城市與城市之間三維的距離位置轉換成二維的資料，由於人類對於高維度的資料難以想像其結構圖形，運用此方法就能讓人們以視覺理解的方式來解讀資料。[3]

多元尺度法主要是利用距離矩陣(其中距離矩陣$d[i,j]$代表位置i與位置j之間的距離)透過一些運算以及double centering(在下一個段落會解釋其作法)的方法把資料平移到質心的位置，由於距離矩陣是對稱矩陣所以作完double centering的矩陣亦是對稱矩陣，所以對這個作完double centering的矩陣作SVD分解可寫成$Z\Sigma Z^T$，把$Z\Sigma Z^T$這個矩陣平方根後取前p項(可寫成Σ的平方根後取前$p \times p$項與Z取前p行後的轉置相乘)即是多元尺度法降到p維後的資料結果。這是$Torgerson$ [15]提出的第一個典型MDS法，它主要的作用是把笛卡兒座標系中從給定的對稱矩陣D來重建矩陣X，其中的關鍵則是應用double centering轉換和SVD分解。

這裡就來介紹double centering的作法，假設X是一個$r \times N$的矩陣，其中N是樣本數的個數、r是資料的維度。令$D = X^T X$、i是$N \times 1$的向量，其中它的每一個元素都是1。Double centering的方法就是把D平移到質心的位置，此時

$$
\begin{aligned}
B &= (X - \tfrac{1}{N} X i i^T)^T (X - \tfrac{1}{N} X i i^T) \\
&= (X^T - \tfrac{1}{N} i i^T X^T)(X - \tfrac{1}{N} X i i^T) \\
&= D - \tfrac{1}{N} D i i^T - \tfrac{1}{N} i i^T D + \tfrac{1}{N^2} i i^T D i i^T
\end{aligned}
$$

$$= D - \bar{D}_r - \bar{D}_c + \bar{D}_g$$

，其中$\bar{D}_r = \frac{1}{N}Dii^T$，$\bar{D}_r$中每列的值是矩陣D沿著列方向作平均，同一列的所有元素值都相等。$\bar{D}_c = \frac{1}{N}ii^TD$，$\bar{D}_c$中每行的值是矩陣$D$沿著行方向作平均，同一行的所有元素值都相等。$\bar{D}_g = \frac{1}{N^2}ii^TDii^T$，$\bar{D}_g$中每個值是整個矩陣D的平均值，所有元素值都相等。把矩陣D轉變成$B = D - \bar{D}_r - \bar{D}_c + \bar{D}_g$這個過程，稱為double centering。若定義$H = I - \frac{1}{N}ii^T$，其中I是$N \times N$的單位矩陣，則矩陣B可寫成

$$B = H^TDH$$

，由於矩陣D、H是對稱矩陣所以矩陣B也會是對稱矩陣，則矩陣B的SVD分解就可以寫成

$$\begin{aligned} B &= (X - \tfrac{1}{N}Xii^T)^T(X - \tfrac{1}{N}Xii^T) \\ &= H^TDH \\ &= Z\Sigma Z^T \\ &= (Z\Sigma^{\frac{1}{2}}P)(P^T\Sigma^{\frac{1}{2}}Z^T) \end{aligned}$$

，其中矩陣P代表一個unitary matrix，不同的矩陣P只是把座標作一個旋轉而已，並不會改變資料的相對位置，為了方便通常在多元尺度法中取$P = I$。而在Σ這個對角矩陣中奇異值會依照大小排序，令$\Sigma_p^{\frac{1}{2}}$為取Σ的平方根後對角線前p項不為零的矩陣以及令Z_p為取Z前p行，則MDS的結果就是

$$\begin{aligned} Y &= \sqrt{B} \\ &= X - \tfrac{1}{N}Xii^T \\ &= \Sigma_p^{\frac{1}{2}}Z_p^T \end{aligned}$$

，其中矩陣$\Sigma_p^{\frac{1}{2}}$是$p \times p$的子矩陣、Z_p^T是$p \times N$的子矩陣、而矩陣Y是矩陣X降成p維後$p \times N$的矩陣。當然在這個分析中D不是距離矩陣，所以推導過程還沒結束，接著要把距離矩陣d創造出來並透過一些運算取代原本的矩陣D，在此我們令d是由矩陣X算出來的距離矩陣(定義$d[i,j] = \sqrt{(X[i] - X[j])^T(X[i] - X[j])}$，其中$X[i]$代表矩陣$X$的第$i$行、也就是矩陣$X$第$i$筆樣本$p$維度的資料)，令$D_1$為距離矩陣$d$自己的點平方(在此所謂的點平方就是把此矩陣中的每個元素值自己平方)再乘以$\frac{-1}{2}$表示為

$$D_1 = \frac{-1}{2}d^{\cdot 2}$$

，現在D_1就是MDS中拿來取代矩陣D的距離矩陣，而且D_1的double centering等於B，$B = H^T DH = H^T D_1 H$ ，依照剛剛的方法把D_1作double centering後再作SVD分解取出矩陣B的平方根後取前r項就是MDS的結果了。但這個方法還是有許多缺點的[5]，像是資料的遺失是無法接受的以及計算量是$O(N^3)$，因此要把這個方法用在大資料上是不適合的。

一般來說多元尺度法最主要的過程是SVD，由於SVD的計算量是$O(N^3)$，所以當矩陣很大時，例如6萬的量現今的個人電腦大概就無法處理了，所以需要一個可以因應大資料計算的方法，就是拆解-合成多元尺度法(Split-and-Combine MDS在此簡稱SC-MDS)。這在$Jengnan\ Tzeng$[17]有提出SC-MDS的理論，它的核心精神就是把資料拆解成很多塊有重疊的子資料，個別經過多元尺度法後轉換成新的座標位置，再把這些有重疊的新作標位置透過旋轉加以合成(由於重疊的座標位置在原本的資料中是一樣的座標，所以可以旋轉加以合成)。接下來，就來介紹如何把新座標透過旋轉接在舊座標上面。

若假設X_1和X_2是從距離矩陣中的兩個有重疊且不相同的子矩陣個別作MDS所得到的矩陣中取重疊部分的行向量(例如：X_1和X_2是距離矩陣取1∼5和3∼7的行與列作MDS得到的矩陣中取第一個矩陣的3∼5行和第二個矩陣的1∼3行)，其中這兩個矩陣是重疊的所以本質上是一樣的資料且彼此距離也沒有改變，因此存在一個仿射映射($affine\ mapping$)可以把X_2的座標映射到X_1的座標。現在令$\bar{X}_1$和$\bar{X}_2$分別是X_1和X_2行向量們的平均值($\bar{X}_1$和$\bar{X}_2$是$p \times 1$的矩陣，其中p是欲降至的維度)，接下來先平移到質心的位置作QR分解就可以使用這些正交化的基底來幫助旋轉，所以$X_1 - \bar{X}_1 i^T = Q_1 R_1$以及$X_2 - \bar{X}_2 i^T = Q_2 R_2$，其中$i$是$r \times 1$的向量它的每一個元素都是1。由於$X_1$和$X_2$本質上是一樣的，所以$R_1$和$R_2$應是一樣的矩陣(由於QR分解不是唯一的，所以對角線上”可能”會差個負號，那就得把對應的Q_2也乘個負號來調整)，調整結束後，我們可以提出

$$Q_1^T(X_1 - \bar{X}_1 i^T) = Q_2^T(X_2 - \bar{X}_2 i^T)$$

，更進一步地我們可以得到

$$X_1 = Q_1 Q_2^T X_2 - Q_1 Q_2^T(\bar{X}_2 i^T) + \bar{X}_1{}^T$$

，因此找出旋轉轉換$U = Q_1 Q_2^T$以及平移轉換$b = -Q_1 Q_2^T(\bar{X}_2 i^T) + \bar{X}_1{}^T$，有了旋轉轉換和平移轉換，我們就有了可以把$X_2$座標映射到$X_1$座標的仿射映射($affine$

$mapping$)。由於每一塊重疊的子資料是個別透過MDS之後再把結果合成出來，所以在主項上面可能會與使用傳統MDS方法算出結果的位置不同，因為在傳統的MDS方法中Component放的順序是以大排到小的，所以在此以PCA的方法再把資料作一個調整，使其結果會與傳統的MDS方法一樣，由於現在資料的維度已經被降到真實維度的大小了，所以在計算時間上並不會太多。

現在就來分析SC-MDS的計算量究竟少的多少，假設資料的樣本數是N個，N_g是子資料的寬度，N_i是重疊的子資料中交集的寬度，當我們把樣本數N拆解成K個子資料時，我們可以得到$KN_g - (K-1)N_i = N$，更近一步可以得到

$$K = \frac{N - N_i}{N_g - N_i}$$

，對於每一個子資料作MDS時，它的計算量會是$O(N_g^3)$，而在重疊的部分是用QR分解，所以它的計算量會是$O(Ni^3)$，因為此資料的真實維度為p，N_i的最小值是p+1，這裡可以假設$N_g = \alpha p$其中α是某些大於2的常數(如果α只比1大一點則有可能使得$\alpha p - p$的值趨近於零)，所以當K個子資料作MDS以及K-1個重疊的部分作QR分解整體的計算量則會是

$$\frac{N-p}{(\alpha-1)p}O(\alpha^3 p^3) + \frac{N-\alpha p}{(\alpha-1)p}O(p^3) \cong O(p^2 N)$$

，當$p << N$時SC-MDS的計算量$O(p^2N)$會比由Morrison et al.在2002年提出的快速MDS方法[13]的計算量$O(\sqrt{N}N)$還要小，然而當p是非常大的數時($p^2 > N$)，SC-MDS在計算量上沒有比較好，但是這個方法提供了一個巨型資料可以運算的方法。而SC-MDS單核心版本的Python Code已放在**附錄A**裡，而其中的過程如圖3.1：

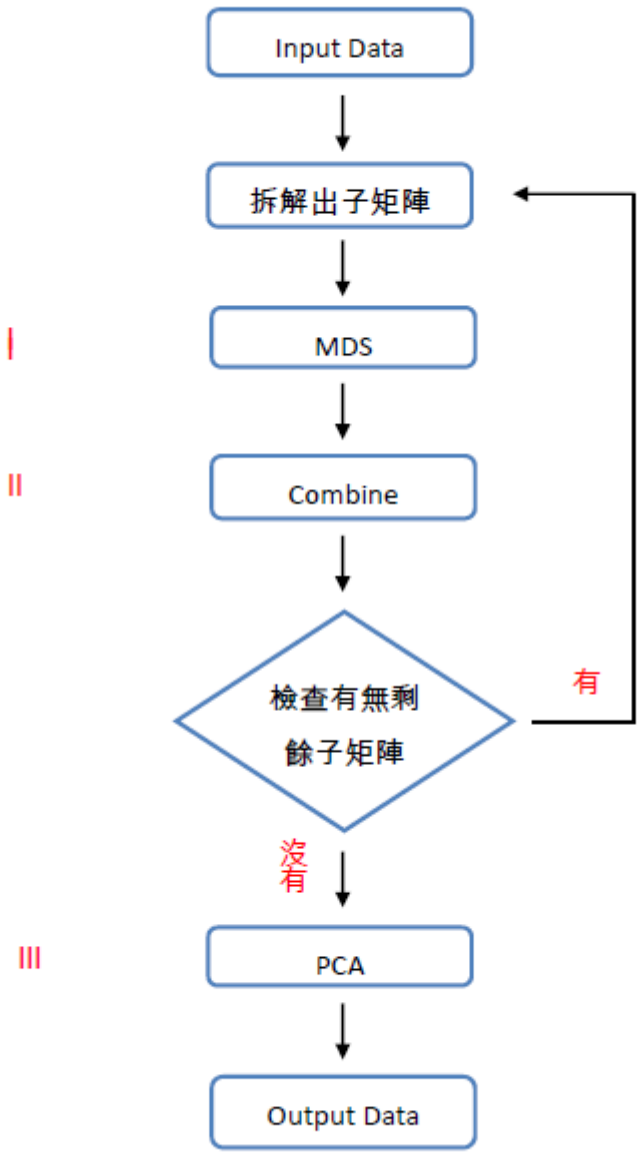


圖 3.1: 單核心SC-MDS流程圖

在圖3.1這個單核心SC-MDS中，這裡我們想算出一般而言哪部分是最占時間的，在此以一個真實維度為50的3000*1000隨機矩陣，接著以Ni = 51、Ng = 2.2*Ni來執行單核心scmds求得的時間個別為I = 1.7811627388000488、II = 0.3949291706085205以及III = 0.03142094612121582，而在別的case中個別花的時間在比例上是沒有很大的差異的，所以這裡可發現到最佔時間的莫過於while迴圈裡的MDS(指的是把資料拆解出子資料後去作MDS的動作)以及Combine(指的是把兩個算出MDS的資料其重疊的部份用QR分解找出它的仿射映射並且接上)，

若拆解的block(指的是資料拆解出的子資料)太大則會發生執行MDS(**附錄A**中的D2X)以及Combine(**附錄A**中的affine_solver)的時間太長而造成速度不快，而若拆解的block太小則會發生block太多使得while迴圈執行太多遍也會造成速度不快。

所以怎麼分配block的大小以及如何把while迴圈裡的東西平行化就成為了加快速度的關鍵，接下來關於操作SC-MDS(**附錄A**中的scmdscale)的參數在這裡作個介紹，這裡先假設真實維度為r、D是距離矩陣、p是我們想估計的真實維度大小、Ni是重疊區塊的大小、Ng是拆解區塊的大小，其中$r \leq p$、$p+1 \leq Ni$、$2*Ni \leq Ng$，這裡參考了$Pei-ChiChen$[6]論文中找出最佳Ni與Ng的參數，所以在接下來的操作也就取Ni = p+1以及Ng = 2.2*Ni讓執行時間達到最快。而這裡也作了一下測試當資料的真實維度為50，大小為3000*1000隨機的矩陣，以p = 50、Ni = p+1的狀況去比較Ng為不同倍數的Ni時的執行時間可發現當Ng = 2.2時間幾乎最快，為了讓答案比較圓滑且公平所以這個例子是透過跑了16次不同的矩陣所畫出來的，由此結果可得到就取Ni = p+1以及Ng = 2.2*Ni為搭配SC-MDS最好的方式。其中橫軸代表Ng取不同倍數的Ni，直軸是執行的時間，如圖3.2：

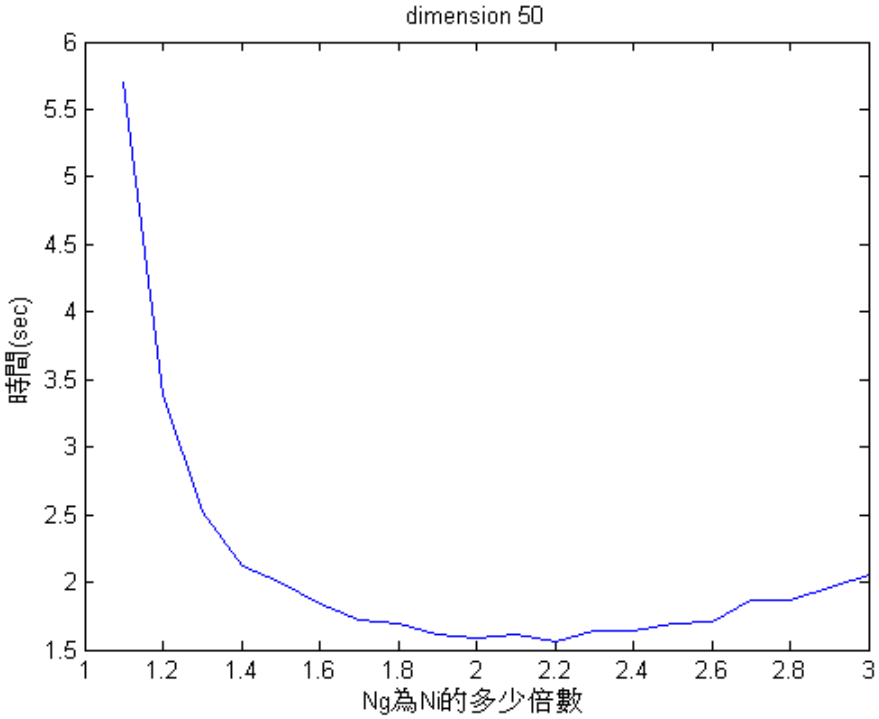


圖 3.2: 找出最佳的Ng

3.2 SCMDS的平行化

在這一節主要會介紹到如何用第二章所學到的方法把第3.1節所提到的SC-MDS平行化，若依照最傳統的MDS法在距離矩陣D中會有很多多餘的資料是不用被拿來計算的如圖3.3的右上角及左下角白色的部分這些都是多餘的資訊，並且整個距離矩陣作MDS時的計算量非常大。而SC-MDS所使用的方法則是拆解出對角線上有重疊的距離矩陣，會大幅縮減MDS時的計算量。由於距離矩陣D在拆解時個別都是獨立的，所以正好可以拿來把MDS平行化並且再把結果放回共享的矩陣中，而交集的部分再透過QR分解把資料合成回去，如圖3.4：

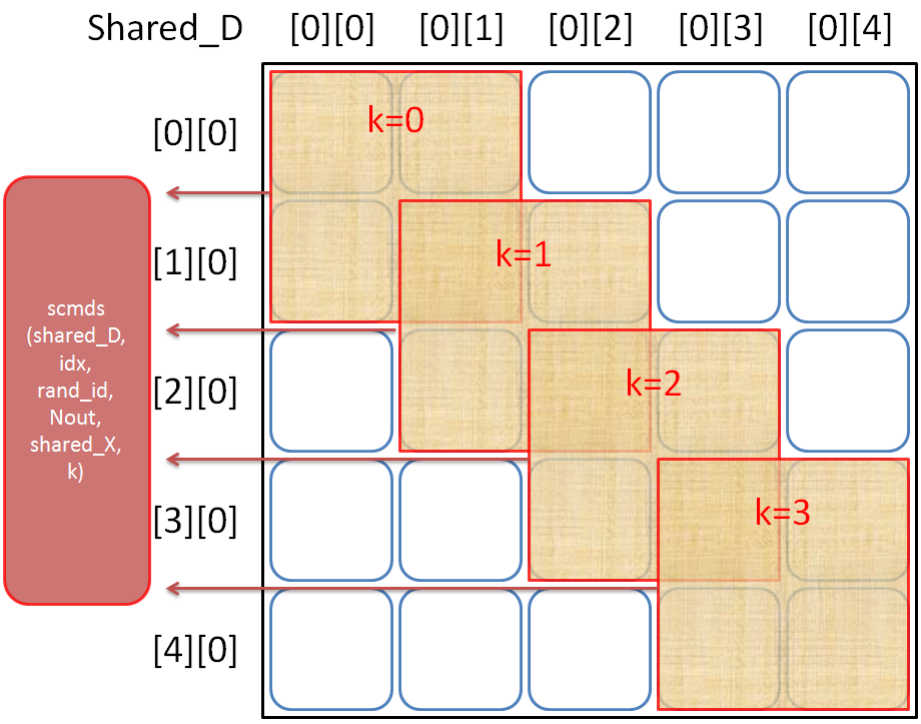


圖 3.3: 拆解平行化示意圖

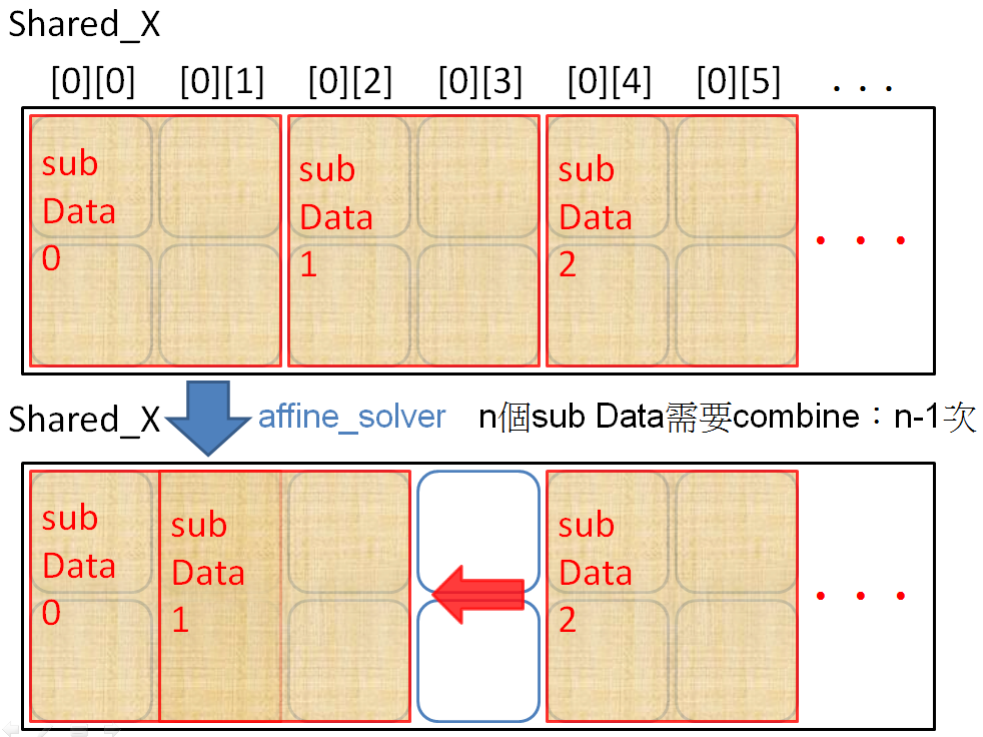


圖 3.4: 合成子資料的示意圖

圖3.4是在解釋圖3.3經過MDS平行化後的狀況，圖3.4的上半部表示的是Shared_X這個共享陣列，它記錄了本來需要大計算量的距離矩陣被拆解成小塊的距離矩陣後透過MDS平行化後的結果，這裡與SC-MDS不一樣的地方則是這裡的每一小塊MDS是透過平行運算的，所以可以把SC-MDS中的這部分再加速，而下半部表示的是這些子資料需要合成的部分，這裡是使用迴圈的方式把兩兩交集的地方經過QR分解把子資料的圖合成回去。

接下來把SC-MDS拆解平行化的過程作一個詳細的介紹，其SC-MDS拆解平行化的程式碼則放在**附錄B**中。有了距離矩陣放入Input Data後，我們得先創造Shared momory，它們是共享的零陣列用來儲存Data與記錄MDS平行化後的結果，接著把值複製到Shared memory上之後就可以開始執行平行化的部分了，從共享的距離矩陣中拆解出子矩陣並且同時透過多個process把MDS平行運算，之後再把結果儲存到共享矩陣中以迴圈的方式把兩兩有交集的部分Combine起來，直到把全部的子資料都Combine完後再執行PCA把主項轉到正確的位置則是SC-MDS拆解平行化的結果了。如圖3.5：

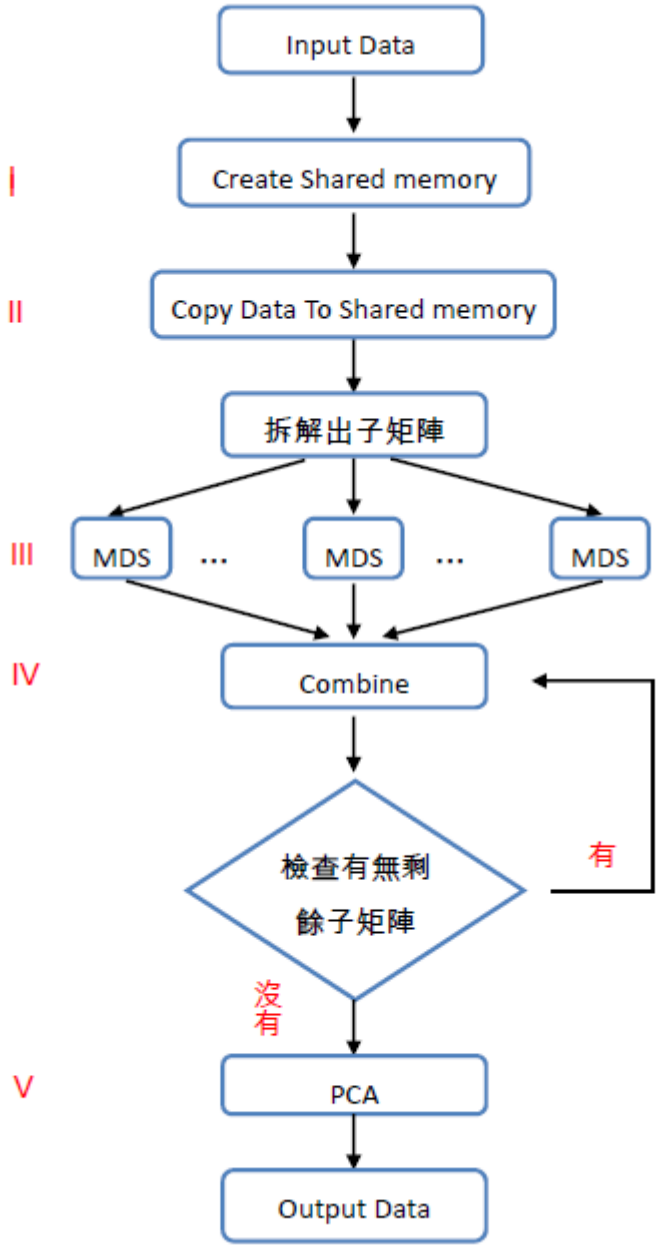


圖 3.5: SC-MDS拆解平行化流程圖

圖3.1與圖3.5兩張流程圖中，圖3.5中的I『Create Shared memory』與II『Copy Data To Shared memory』是比圖3.1多出來的，以及III『MDS』的部分也從單核心被拆解成平行化的版本，其餘的部分在結構上是一樣的，所以在這裡我們期待I與II的執行時間是短的，並且如果III的部分比單核心『MDS』的執行時間快很多時，SC-MDS的平行化就會有其利基。

第 四 章　實驗結果

首先，本篇先提供所使用的硬體當參考；處理器：AMD Opteron(tm) Processor 6128 2GHz、核心數：8、記憶體：64GB、作業系統Ubuntu 64位元12.04版、Python：2.7.1版。本章以SC-MDS以及SC-MDS拆解平行化為主要探討對象，接著把多核心的控制帶入SC-MDS與其平行化來比較在不同核心數時對時間的差異，接著再找出每個階段的執行時間對於核心的提升到底有沒有提升，以及其加速最多的地方並且比較其效率。

4.1　SC-MDS與其平行化在多核心控制下之比較

因為SC-MDS是應用大資料的計算所以本篇所測試的資料量會比較大，在此估計三種不同的真實維度個別為資料一：50維度、資料二：100維度以及資料三：150維度的大資料在不同時的核心數底下對於SC-MDS與其平行化來進行測試並比較其效率。

在此先以資料一：50維度來當範例，這裡要測試的程式分別為SC-MDS單核心版本、2核心底下的SC-MDS平行化、4核心底下的SC-MDS平行化、8核心底下的SC-MDS平行化以及16核心底下的SC-MDS平行化在樣本數分別在3000、6000、$\cdots$、15000這五種csae時所執行出來的時間提升狀況，把資料隨機創造出來並計算出距離矩陣後，丟給SC-MDS單核心版本、2核心底下的SC-MDS平行化、4核心底下的SC-MDS平行化、8核心底下的SC-MDS平行化以及16核心底下的SC-MDS平行化來計算執行時間，其中真實維度p = 50、重疊區塊的大小(Ni)以及拆解區塊的大小(Ng)採用3.1節所提出最佳的參數為Ni = p+1以及Ng = 2.2*Ni。範例如下：

SC-MDS的範例 樣本數為3000時

```
01| >>> import scmds as sc
  # 引用附錄A、B所存成的scmds.py檔
```

```
02| >>> import affinity
  # 引入affinity套件
03| >>> A = sc.np.random.random((3000,50))
04| >>> B = sc.np.random.random((50,1000))
05| >>> C = sc.np.dot(A,B)
  # 這裡的C矩陣是一個真實維度為50的矩陣，矩陣大小是3000*1000，其中樣本數為3000、維度為1000
06| >>> D = sc.pdist(C)
  # 此為算出彼此的距離
07| >>> D = sc.squareform(D[1])
  # 此為把剛剛算出的距離放入距離矩陣內
08| >>> Y = sc.scmdscale(D,50,int(sc.np.floor(2.2*(50+1))),50+1,0)
  # 執行SC-MDS單核心版，p = 50、Ng = 2.2*Ni、Ni = p+1
09| >>> Y[0]
  # 顯示執行時間
    2.1422
  # 2.1422(sec)
10| >>> affinity.set_process_affinity_mask(0,2**2-1)
  # 控制核心使用數目為2顆
11| >>> Y1 = sc.pscmdscale(D,50,int(sc.np.floor(2.2*(50+1))),50+1,0)
  # 執行2核心底下的SC-MDS平行化，p = 50、Ng = 2.2*Ni、Ni = p+1
12| >>> Y1[0]
  # 顯示執行時間
    1.6415
  # 1.6415(sec)
13| >>> affinity.set_process_affinity_mask(0,2**4-1)
  # 控制核心使用數目為4顆
14| >>> Y2 = sc.pscmdscale(D,50,int(sc.np.floor(2.2*(50+1))),50+1,0)
  # 執行4核心底下的SC-MDS平行化，p = 50、Ng = 2.2*Ni、Ni = p+1
15| >>> Y2[0]
  # 顯示執行時間
```

```
    1.0969
  # 1.0969(sec)
16| >>> affinity.set_process_affinity_mask(0,2**8-1)
  # 控制核心使用數目為8顆
17| >>> Y3 = sc.pscmdscale(D,50,int(sc.np.floor(2.2*(50+1))),50+1,0)
  # 執行8核心底下的SC-MDS平行化，p = 50、Ng = 2.2*Ni、Ni = p+1
18| >>> Y3[0]
  # 顯示執行時間
    0.8424
  # 0.8424(sec)
19| >>> affinity.set_process_affinity_mask(0,2**16-1)
  # 控制核心使用數目為16顆
20| >>> Y4 = sc.pscmdscale(D,50,int(sc.np.floor(2.2*(50+1))),50+1,0)
  # 執行16核心底下的SC-MDS平行化，p = 50、Ng = 2.2*Ni、Ni = p+1
21| >>> Y4[0]
  # 顯示執行時間
    0.7444
  # 0.7444(sec)
```

依照上述範例的方式，當樣本數為3000時，把SC-MDS單核心版本、2核心底下的SC-MDS平行化、4核心底下的SC-MDS平行化、8核心底下的SC-MDS平行化以及16核心底下的SC-MDS平行化五種方法所得到的時間存起來則是樣本數為為3000的結果。接著想觀察在別的樣本數下不同核心數對於執行時間的比值到底有沒有進步，所以依序設定樣本數為3000、6000、$\cdots$、15000共五種case執行以上的步驟，由於資料是隨機產生的所以為了讓資料更趨於穩定，因此在這裡的資料都會再重新創造與執行十五次，並把十六次的執行時間作一個平均值，而這只是資料一：真實維度為50的例子，依序把資料二：真實維度為100與資料三：真實維度為150也拿來測試，下圖就是把執行結果畫出來，其中橫軸是樣本數的大小，直軸是執行的時間，如圖4.1、圖4.2、圖4.3：

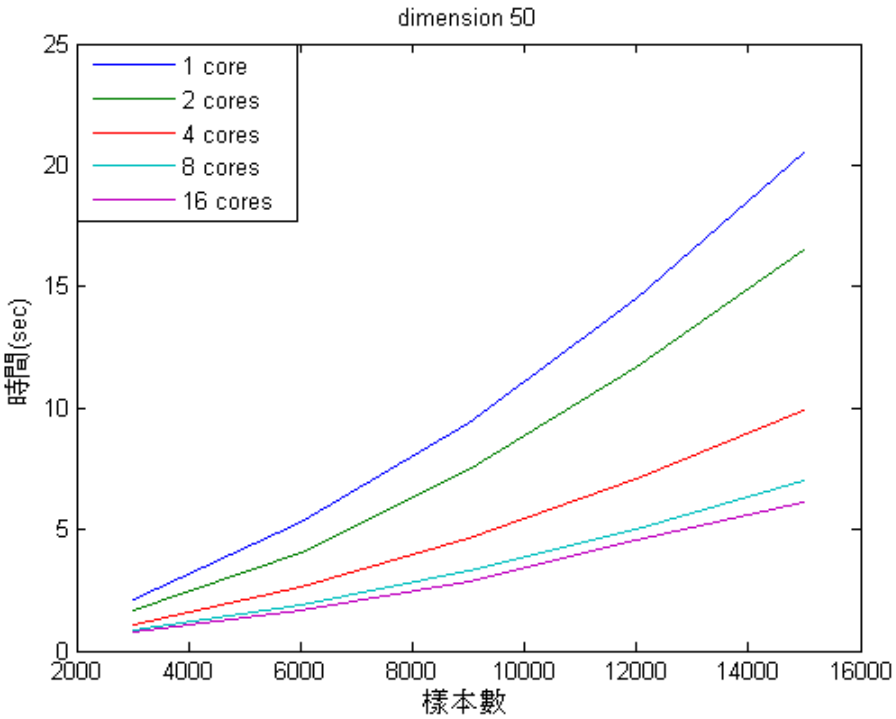


圖 4.1: SC-MDS與其平行化的執行時間比較圖(真實維度為50)

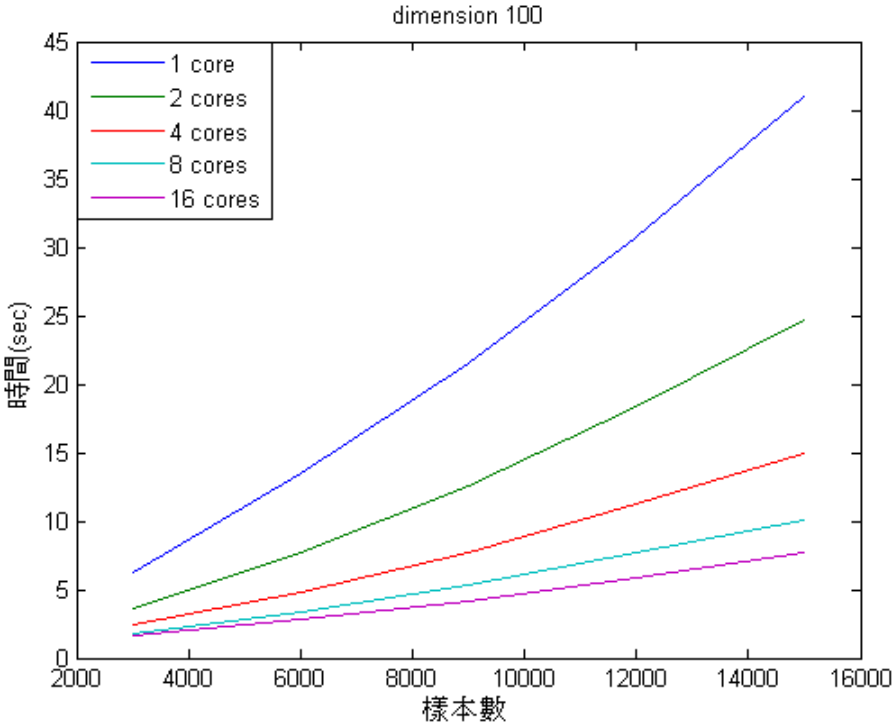


圖 4.2: SC-MDS與其平行化的執行時間比較圖(真實維度為100)

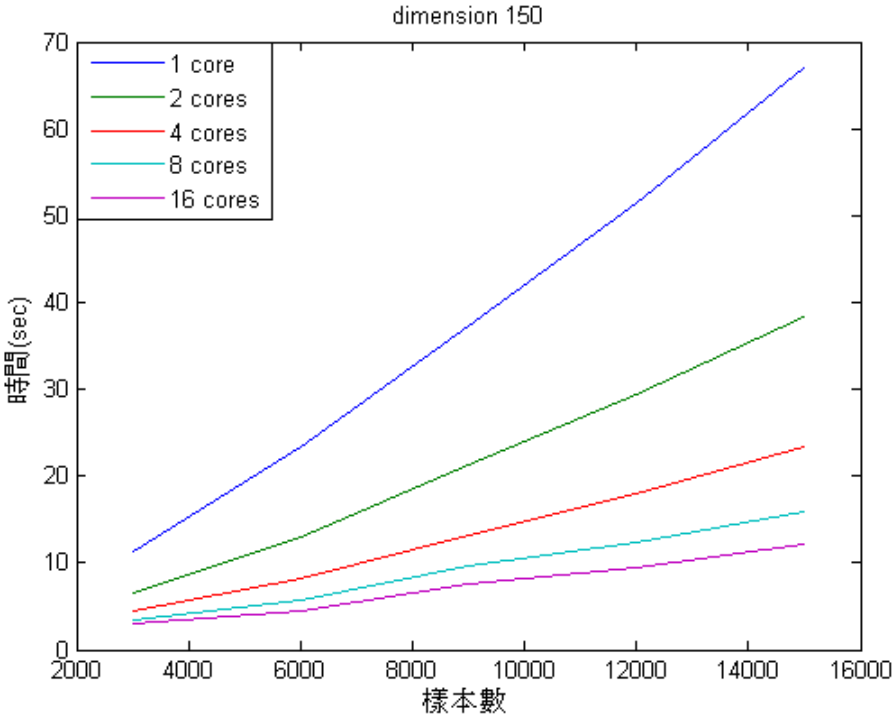


圖 4.3: SC-MDS與其平行化的執行時間比較圖(真實維度為150)

從圖4.1、圖4.2、圖4.3中可觀察到雖然SC-MDS單核心版本的執行時間並不是2核心底下的SC-MDS平行化執行時間的兩倍，以及SC-MDS單核心版本的執行時間與其它倍數核心底下的SC-MDS平行化執行時間的比值(這裡的比值代表的是SC-MDS單核心版本的執行時間除以別的核心數底下的執行時間)也不是核心數的倍率，核心的倍數雖然並沒有讓時間也達到一樣倍數的縮短，不過在執行時間上還是少了許多，接下來就來比較這些比值究竟是幾倍，如表4.1、表4.2、表4.3：

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	2.1422	1.6415	1.0969	0.8424	0.7444	時間
	1	**1.3050**	**1.9530**	**2.5429**	**2.8778**	比值
6000	5.3217	4.0766	2.6188	1.9194	1.6515	時間
	1	**1.3054**	**2.0321**	**2.7726**	**3.2223**	比值
9000	9.3574	7.4448	4.6341	3.3021	2.8673	時間
	1	**1.2569**	**2.0193**	**2.8338**	**3.2635**	比值
12000	14.5039	11.6666	7.1063	5.0306	4.5349	時間
	1	**1.2432**	**2.0410**	**2.8831**	**3.1983**	比值
15000	20.5157	16.5000	9.9324	6.9757	6.1258	時間
	1	**1.2434**	**2.0655**	**2.9410**	**3.3490**	比值

表 4.1: SC-MDS與其平行化執行時間的比值(真實維度為50)

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	6.2910	3.5839	2.4306	1.8347	1.5921	時間
	1	**1.7553**	**2.5883**	**3.4289**	**3.9514**	比值
6000	13.4576	7.6506	4.7449	3.4064	2.7632	時間
	1	**1.7590**	**2.8362**	**3.9506**	**4.8702**	比值
9000	21.5460	12.5408	7.7334	5.3402	4.1564	時間
	1	**1.7181**	**2.7861**	**4.0347**	**5.1838**	比值

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
12000	30.6964	18.4055	11.2386	7.6622	5.9102	時間
	1	**1.6678**	**2.7313**	**4.0062**	**5.1938**	比值
15000	40.9411	24.6853	14.9504	10.0390	7.6501	時間
	1	**1.6585**	**2.7385**	**4.0782**	**5.3517**	比值

表 4.2: SC-MDS與其平行化執行時間的比值(真實維度為100)

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	11.2975	6.6218	4.5232	3.4546	2.9370	時間
	1	**1.7061**	**2.4977**	**3.2703**	**3.8467**	比值
6000	23.3318	13.0014	8.1566	5.7194	4.5551	時間
	1	**1.7946**	**2.8605**	**4.0794**	**5.1222**	比值
9000	37.3263	21.2125	13.2836	9.5911	7.6429	時間
	1	**1.7596**	**2.8100**	**3.8918**	**4.8838**	比值
12000	51.3441	29.4349	17.9716	12.2554	9.3488	時間
	1	**1.7443**	**2.8570**	**4.1895**	**5.4921**	比值
15000	67.0523	38.4071	23.3575	15.9491	12.0683	時間
	1	**1.7458**	**2.8707**	**4.2041**	**5.5561**	比值

表 4.3: SC-MDS與其平行化執行時間的比值(真實維度為150)

就上述表4.1、表4.2、表4.3可觀察到雖然在核心數不高時樣本數的提升不一定能提升比值，但在我們的測試中比值最少也有1.2432(表4.1的2核心比值)所以執行時間還是優於SC-MDS單核心版本的執行時間，而除了在核心數低的狀況下我們還發現到在『固定的核心數底下樣本數的提升』以及『固定的樣本數底下核心數的提升』都會使比值提高，如果這個理論是正確的，比值的最大值應該會發生在核心數與樣本數最高的時候，而這與我們的測試結果一致，這是非常好的事實因為當樣本數越大時多核心的平行化效率就會越好了。

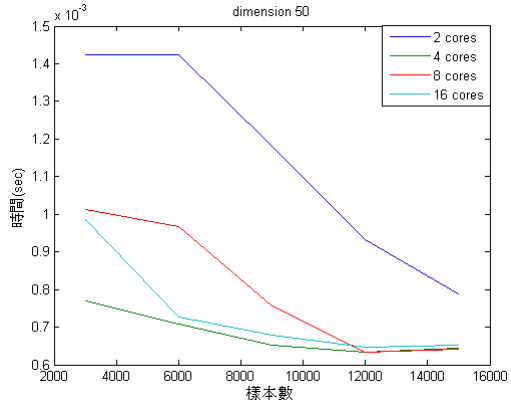
在測試中比值的最大值為5.5561(在表4.3的16核心比值)，這到底是多核心的操作對於SC-MDS中哪個階段造成的影響呢，下一節會以這個方向為主要目的並找出其最有影響的階段。

4.2 多核心的操作對於SC-MDS的平行化在各個階段的影響

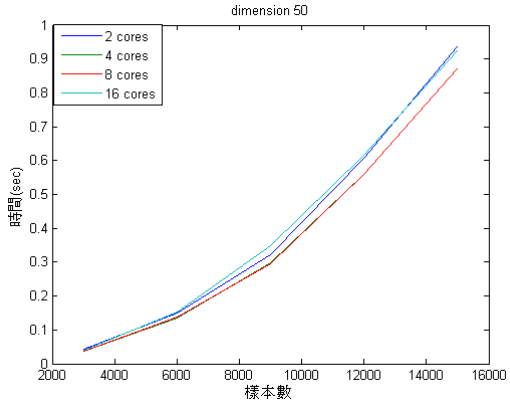
根據現在已知多核心的操作對於平行化已有顯著的成效，但是SC-MDS平行化版本中有很多階段在多核心的運算下效果是一樣的，所以為了公平起見在這裡會把SC-MDS平行化各個階段的執行時間都記錄下來藉以了解在各階段的時間狀況。因此這裡的測試也與4.1節採用相同的測試條件。

SC-MDS的平行化總共分為I.Create Shared memory、 II.Copy Data To Shared memory、III.MDS、IV.Combine以及V.PCA共五個階段(如同3.2節所提及的五個階段)。而這一節的測試則把這五種階段以及全部所花的時間VI.Total time作一個比較。

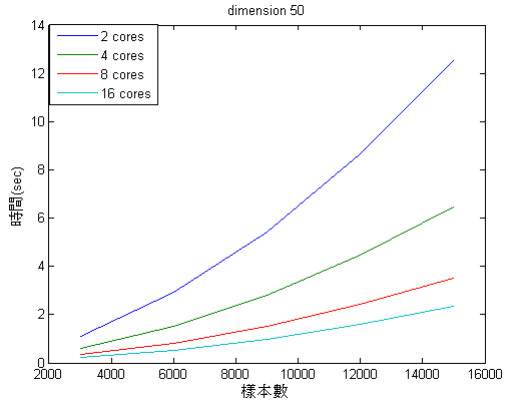
在此以三種不同的真實維度個別為資料一：50維度、資料二：100維度以及資料三：150維度的大資料以重疊區塊的大小Ni = p+1以及拆解區塊的大小Ng = 2.2*Ni(採用3.1節所提出最佳的參數)在樣本數3000、6000、· · ·、15000共五種case在不同的核心使用數下測試各階段的執行時間以及比較其效率。而這一節資料的產生方式和4.1節相同，下圖就是把執行結果畫出來，其中橫軸是樣本數的大小，直軸是執行的時間，如圖4.4、圖4.5、圖4.6：



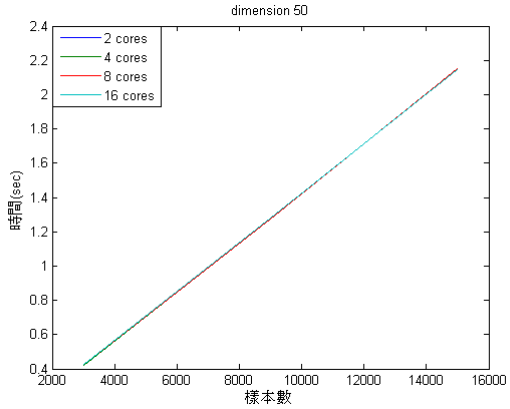
(a) I.Create Shared memory



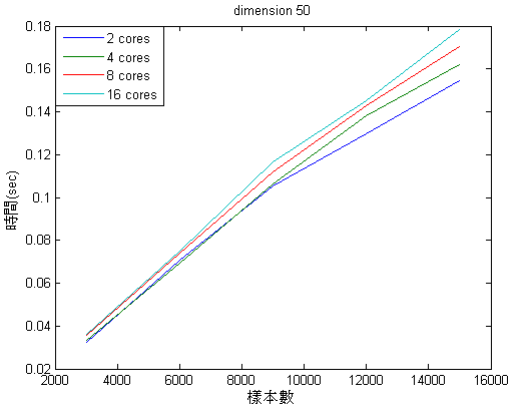
(b) II.Copy Data To Shared memory



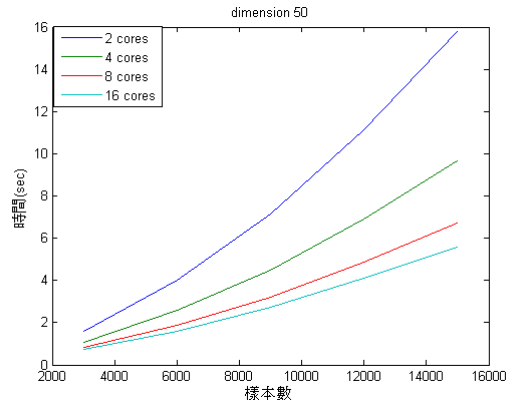
(c) III.MDS



(d) IV.Combine

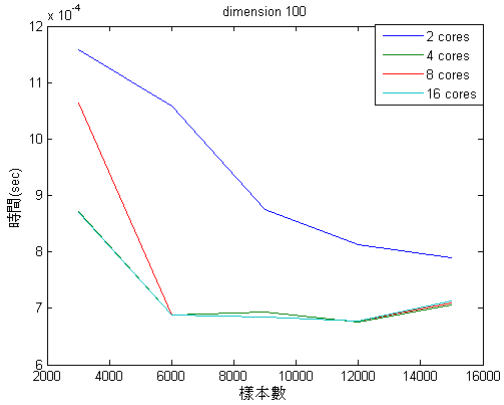


(e) V.PCA

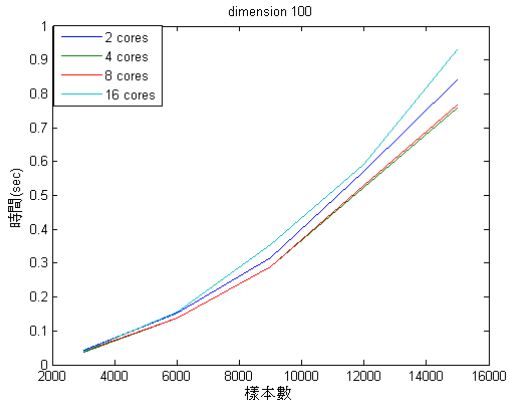


(f) VI.Total time

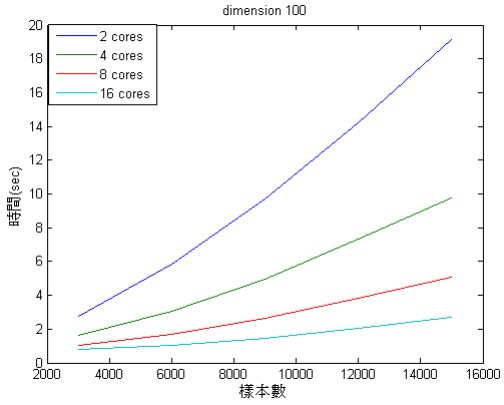
圖 4.4: SC-MDS的平行化在各個階段的執行時間比較圖(真實維度為50)



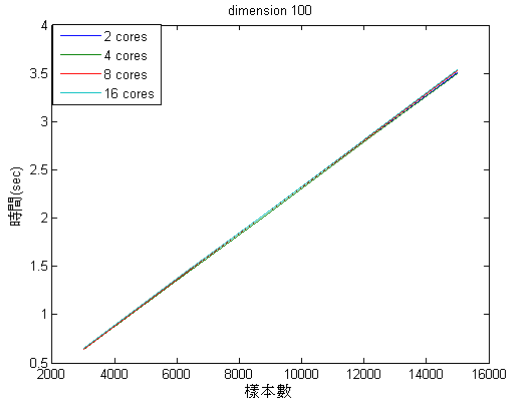
(a) I.Create Shared memory



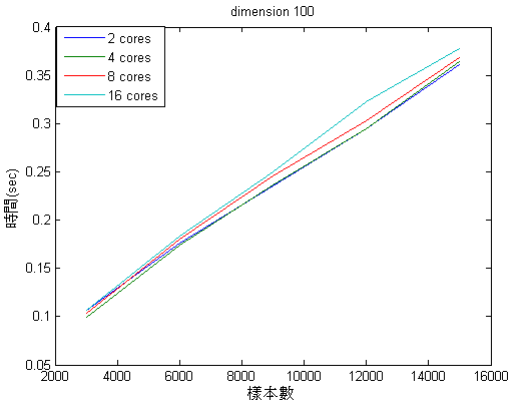
(b) II.Copy Data To Shared memory



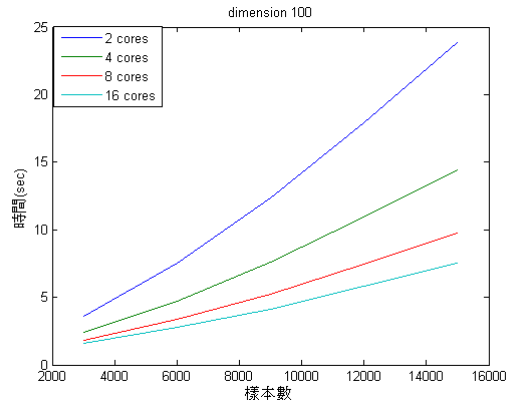
(c) III.MDS



(d) IV.Combine

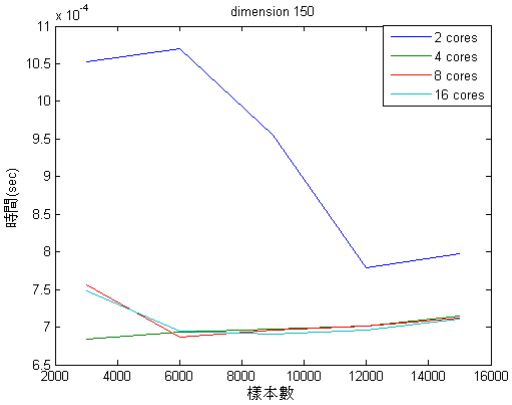


(e) V.PCA

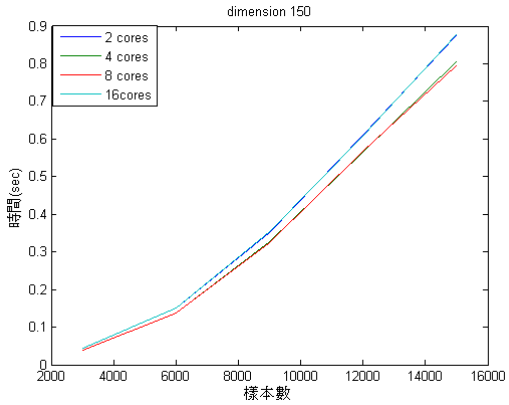


(f) VI.Total time

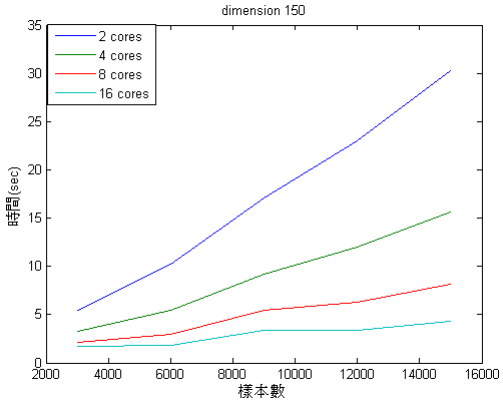
圖 4.5: SC-MDS的平行化在各個階段的執行時間比較圖(真實維度為100)



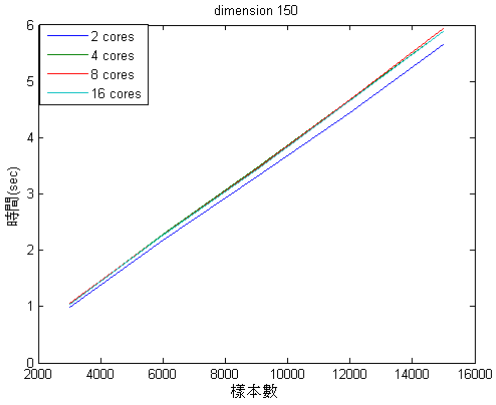
(a) I.Create Shared memory



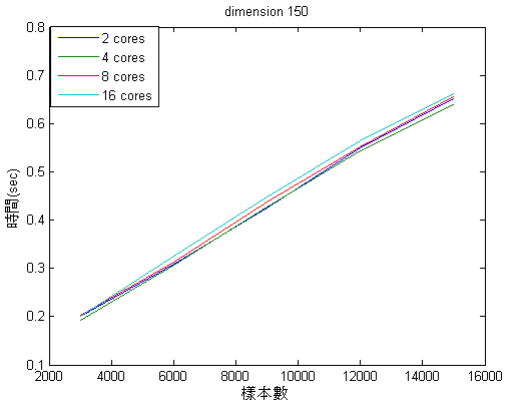
(b) II.Copy Data To Shared memory



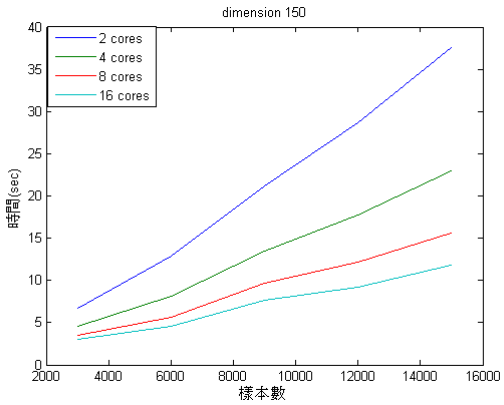
(c) III.MDS



(d) IV.Combine



(e) V.PCA



(f) VI.Total time

圖 4.6: SC-MDS的平行化在各個階段的執行時間比較圖(真實維度為150)

由圖4.4、圖4.5、圖4.6中可發現到它們的子圖(a)I.Create Shared memory在核心使用數不同時彼此的執行時間相差很小因為它的單位是$10^{-4} \sim 10^{-3}$，而他們的子圖(b) II.Copy Data To Shared memory、(d)IV.Combine以及(e)V.PCA在核心使用數不同時執行時間會隨樣本數增加但彼此的執行時間在不同核心上沒有明顯的改變，可見核心數的多寡和上述幾個階段都沒有太大的關係，而剩下的子圖(c)III.MDS中則發現在核心使用數不同時彼此的執行時間相差很大並且它們的曲線圖與(f)VI.Total time有直接的相關，所以這裡可以得到兩個小結論而第一個結論是：從子圖(b) II.Copy Data To Shared memory與(c)III.MDS可發現到當核心使用數到達16核時III.MDS的執行時間已經被縮短到幾乎不到3秒而II.Copy Data To Shared memory卻不會因為核心數增多而降低執行時間，還需要花0.8~0.9秒，所以當核心使用數再增加到某個數字時就會發生II.Copy Data To Shared memory的執行時間會超過III.MDS的執行時間了，可以得到『核心使用數在提升平行化效率時有個最大值，意即超過這個最大值，多核心就不划算了』。第二個結論是：從子圖(c)III.MDS 與(d)IV.Combine可發現到當核心使用數到達16核時IV.Combine的執行時間已經超過III.MDS的執行時間了，所以就算核心數再增加多核心加速的效率還是會被IV.Combine所侷限住。

4.3 多核心的操作中MDS平行化的效能比

透過4.2節的討論，核心數使用的多寡和4.2節中III.MDS階段加速的效率才是最息息相關的，其它階段加速的效率與多核心的操作似乎沒有直接的影響，為了客觀地比較多核心平行化與單核心的效能，這節所採取的方式是比較SC-MDS單核心版本中的MDS階段以及在2核心、4核心、8核心以及16核心底下SC-MDS平行化中的MDS階段。因此這裡的測試也與4.1、4.2節採用相同的測試條件。

在此以三種不同的真實維度個別為資料一：50維度、資料二：100維度以及資料三：150維度的大資料以重疊區塊的大小Ni = p+1以及拆解區塊的大小Ng = 2.2*Ni(採用3.1節所提出最佳的參數)在樣本數3000、6000、$\cdots$、15000共五種case在不同的核心使用數下測試MDS階段的執行時間以及比較其效率。而這一節資料的產生方式和4.1節相同，下圖就是把執行結果畫出來，其中橫軸是樣本數的大小，直軸是執行的時間，如圖4.7、圖4.8、圖4.9：

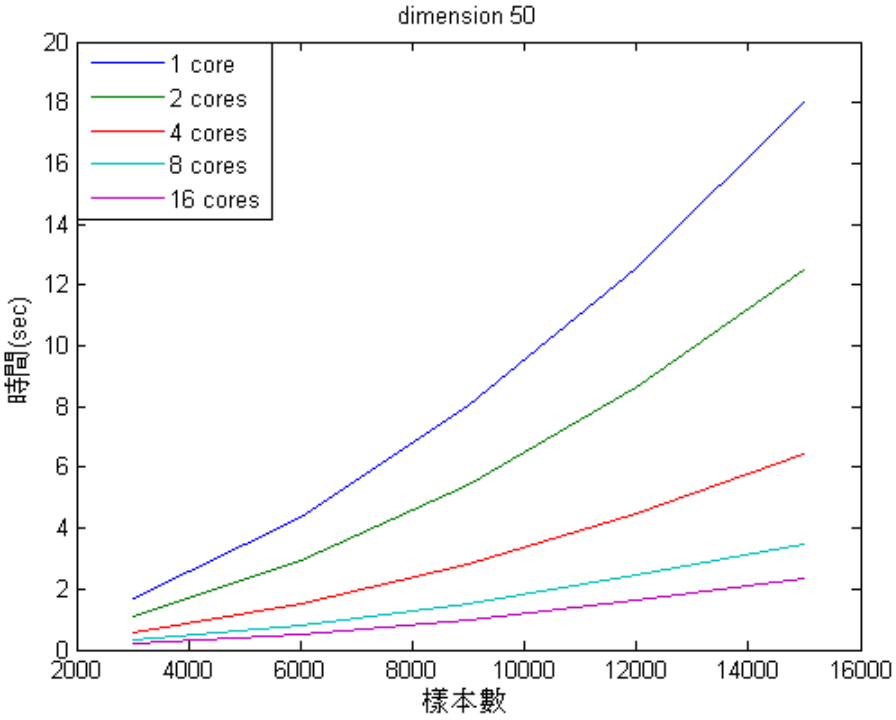


圖 4.7: SC-MDS與其平行化在MDS階段的執行時間比較圖(真實維度為50)

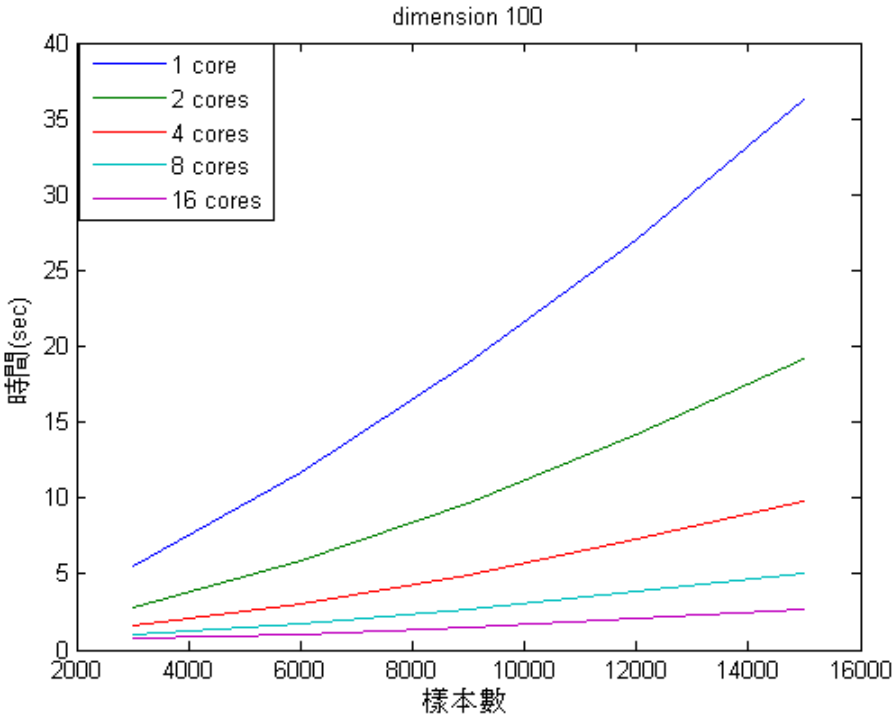


圖 4.8: SC-MDS與其平行化在MDS階段的執行時間比較圖(真實維度為100)

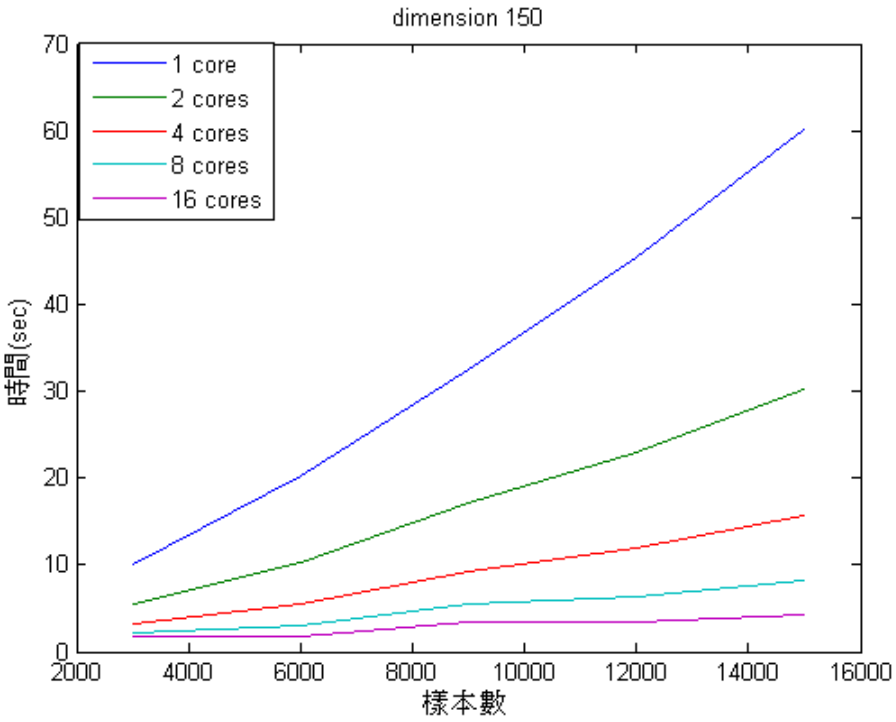


圖 4.9: SC-MDS與其平行化在MDS階段的執行時間比較圖(真實維度為150)

從圖4.7、圖4.8、圖4.9中可觀察到SC-MDS與其平行化在MDS階段的執行時間的比值(這裡的比值代表的是SC-MDS單核心版本MDS階段的執行時間除以別的核心數底下MDS階段的執行時間)似乎與核心使用數的倍數相差不遠了，下來就來比較這些比值究竟是幾倍，如表4.4、表4.5、表4.6：

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	1.7061	1.1089	0.5865	0.3330	0.2221	時間
	1	**1.5386**	**2.9090**	**5.1234**	**7.6817**	比值
6000	4.3671	2.9114	1.5147	0.8225	0.5184	時間
	1	**1.5000**	**2.8831**	**5.3095**	**8.4242**	比值
9000	8.0410	5.4152	2.8160	1.5187	0.9854	時間
	1	**1.4849**	**2.8555**	**5.2947**	**8.1601**	比值

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
12000	12.5504	8.6574	4.4802	2.4372	1.6073	時間
	1	**1.4497**	**2.8013**	**5.1495**	**7.8084**	比值
15000	18.0065	12.5113	6.4580	3.4986	2.3291	時間
	1	**1.4392**	**2.7882**	**5.1468**	**7.7311**	比值

表 4.4: SC-MDS與其平行化在MDS階段執行時間的比值(真實維度為50)

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	5.5207	2.7760	1.6284	1.0172	0.7829	時間
	1	**1.9888**	**3.3902**	**5.4272**	**7.0519**	比值
6000	11.7333	5.8686	3.0312	1.7012	1.0550	時間
	1	**1.9993**	**3.8708**	**6.8970**	**11.1213**	比值
9000	18.9458	9.6866	4.9831	2.6254	1.4400	時間
	1	**1.9559**	**3.8020**	**7.2164**	**13.1570**	比值
12000	27.0553	14.2258	7.3021	3.8420	2.0726	時間
	1	**1.9018**	**3.7051**	**7.0419**	**13.0540**	比值
15000	36.2851	19.1267	9.7574	5.0889	2.6765	時間
	1	**1.8971**	**3.7187**	**7.1302**	**13.5567**	比值

表 4.5: SC-MDS與其平行化在MDS階段執行時間的比值(真實維度為100)

對應樣本數(時間和比值)\核心數	1	2	4	8	16	
3000	9.9896	5.4291	3.2899	2.1606	1.6823	時間
	1	**1.8400**	**3.0364**	**4.6235**	**5.9382**	比值
6000	20.3222	10.2607	5.4188	2.9336	1.7700	時間
	1	**1.9806**	**3.7503**	**6.9275**	**11.4812**	比值
9000	32.4566	17.0361	9.2043	5.4336	3.3713	時間
	1	**1.9052**	**3.5262**	**5.9733**	**9.6273**	比值
12000	45.3140	23.0451	11.9607	6.2891	3.3675	時間
	1	**1.9663**	**3.7886**	**7.2051**	**13.4561**	比值
15000	60.1014	30.2908	15.5924	8.1968	4.3285	時間
	1	**1.9842**	**3.8545**	**7.3323**	**13.8851**	比值

表 4.6: SC-MDS與其平行化在MDS階段執行時間的比值(真實維度為150)

就上述表4.4、表4.5、表4.6可發現到原來SC-MDS平行化中的MDS階段可以被多核心加速到最快13.8851倍(表4.6的16核心比值)，最慢也有1.4392倍(表4.3的2核心比值)所以執行時間還是優於SC-MDS單核心版本MDS階段的執行時間，而這裡也觀察到一點與4.1節相同的地方就是除了在核心數低的狀況下我們還發現到在『固定的核心數底下樣本數的提升』以及『固定的樣本數底下核心數的提升』都會使比值提高，比值的最大值會發生在核心數與樣本數最高的時候，因此當樣本數越大時多核心的平行化效率就會越好了並且若只討論平行化的階段還可以使比值達到接近核心使用數的數量。

第 五 章　結論

由本論文的實驗結果可得到以下結論：

1. 在4.2節所提到的II.Copy Data To Shared memory在資料的派送上會隨著樣本數的提升而增加執行時間。

2. 如結論1在資料的派送上會有個固定的執行時間是多核心所不能縮短的，所以當核心使用數越高時，派送資料的時間就會佔越大部分使得多核心下的SC-MDS平行化效率降低，意即核心使用數在提升平行化效率時有個最大值，超過這個最大值，多核心就不划算了。

3. 在16核底下的SC-MDS平行化時，在4.2節所提到的IV.Combine的執行時間已經超過III.MDS的執行時間了，所以就算核心數再增加多核心加速的效率還是會被IV.Combine所侷限住。

SC-MDS的方法已經把傳統的MDS方法大幅改進而SC-MDS平行化又再次的提高其效率，若要再提升SC-MDS平行化的效率就得從IV.Combine的階段著手，由於在執行平行化時資料都已先被存入Shared memory所以已經省下了資料派送的時間，若能寫出其平行化的程式可望能把效率作再次的提升。

參 考 文 獻

[1] David Griffiths、Paul Barry. 深入淺出程式設計. 歐萊禮, 2011.

[2] Paul Barry. 深入淺出 *Python*. 歐萊禮, 2011.

[3] Ingwer Borg and Patrick J. F. Groenen. *Modern multidimensional scaling*. Springer Series in Statistics. Springer, New York, second edition, 2005. Theory and applications.

[4] TOIBE Software BV. Tiobe programming community index, 2013. [online] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[5] Matthew Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 127–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[6] Pei-Chi Chen. Optimal grouping and missing data handling for split-and-combine multidimensional scaling. 2008.

[7] Michael A. A. Cox and Trevor F. Cox. Multidimensional scaling. In *Handbook of Data Visualization*, Springer Handbooks Comp.Statistics, pages 315–347. Springer Berlin Heidelberg, 2008.

[8] Pearu Peterson Eric Jones, Travis Oliphant et al. Open source scientific tools for python, 2001. [online] `http://www.scipy.org/`.

[9] Python Software Foundation. About python, 2005. [online] `http://www.python.org/about/`.

[10] Python Software Foundation. affinity 0.1.0, 2005. [online] `https://pypi.python.org/pypi/affinity`.

[11] Python Software Foundation. Process-based "threading" interface, 2005. [online] `http://docs.python.org/2/library/multiprocessing.html`.

[12] Swaroop C H. Python入門, 2013. [online] http://files.swaroopch.com/python/byte_of_python.pdf.

[13] Alistair Morrison, Greg Ross, and Matthew Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2:68–77, 2003.

[14] Mark Pilgrim. Dive into python, 2004. [online] http://www.diveintopython.net/toc/index.html.

[15] Warren S. Torgerson. Multidimensional scaling. I. Theory and method. *Psychometrika*, 17:401–419, 1952.

[16] Jengnan Tzeng. Python入門, 2009. [online] http://dl.dropboxusercontent.com/u/2688690/python_note.html.

[17] Jengnan Tzeng. Split-and-combine singular value decomposition for large-scale matrix. *J. Appl. Math.*, pages Art. ID 683053, 8, 2013.

[18] Guido van Rossum. Python tutorial, 2008. [online] http://docs.python.org/2.5/tut/tut.html.

附錄A：SC-MDS單核心版本的code

SC-MDS單核心版本的code

```
# scmds.py
001| '''
002| 001~022為目錄
003| 023~032引入套件
004| 033~038是副函式double
005| 039~049是副函式zero_sum
006| 050~059是副函式col_mean
007| 060~069是副函式row_mean
008| 070~092是副函式affine_solver
009| 093~109是副函式sort_idx
010| 110~128是副函式pdist
011| 129~152是副函式squareform
012| 153~174是副函式D2X
013| 175~193是副函式kron
014| 194~238是副函式scmdscale
015| 239~255是副函式shared_array
016| 256~264是副函式array2shared
017| 265~329是副函式pscmdscale
018| 330~434是副函式pscmdscale1
019| 033~193為宣告一些方便scmdscale計算的函數
020| 239~264 為宣告一些方便平行化所需要的函數
021| '''
022| # 上面的'''    '''表示的是大篇幅的註解，而一個#表示的是小篇幅的註解
023| import multiprocessing as mp
024| import ctypes
025| import numpy as np
026| import time
```

```
import os
from numpy import linalg as la
from numpy import random as ra
import operator
from numpy.lib import scimath as cnp
from scipy.linalg import orth as orth
```

先引入一些會用到的套件

```
def double(A):
  """
  Make the matrix becomes double type
  Syntax: A = double(A)
  """
  return 1.0*A
```

因為python初始的運算是以整數來運算，所以要除到小數後幾位的話就要用這個副函式把數值轉成浮點數。

```
def zero_sum(A):
  """
  Make the column mean of A to be zero. If row of A is data,
  zero_sum move the data set becomes zero center of mass.
  Syntax: B = zero_sum(A)
  """
  m,n = A.shape
  A = double(A)
  for i in range(n):
    A[:,i] = A[:,i] - np.mean(A[:,i])
  return A
```

計算直和，就是把矩陣A的每一行減去那一行的平均

```
def col_mean(A):
  """
```

```
    Compute the column mean of matrix.
    Syntax: cm = col_mean(A)
    """
    m,n = A.shape
    cm = np.zeros(n)
    for i in range(n):
      cm[i] = np.mean(A[:,i])
    return cm
# 計算行平均

def row_mean(A):
    """
    Compute the row mean of matrix.
    Syntax: rm = row_mean(A)
    """
    m,n = A.shape
    rm = np.zeros(m)
    for i in range(m):
      rm[i] = np.mean(A[i,:])
    return rm
# 計算列平均

def affine_solver(X,Y):
     """
     Return a affine mapping Y = U*X+b, U*U.T = I
     Syntax: (U,b) = affine_solver(X,Y)
     """
     m1,n1 = X.shape
     m2,n2 = Y.shape
     if m1<>m2 or n1<>n2:
       print 'size of X and Y must be the same'
       return
```

```
    cmx = col_mean(X)
    cmy = col_mean(Y)
    ZX = zero_sum(X)
    ZY = zero_sum(Y)
    (QX,RX) = la.qr(ZX.T)
    (QY,RY) = la.qr(ZY.T)
    m,n = QX.shape
    for p in range(n):
      if np.sign(RX[p,p])<>np.sign(RY[p,p]):
        QY[:,p] = -QY[:,p]
    U = np.dot(QY,QX.T)
    b = cmy-np.dot(U,cmx)
    return U,b
```

在此的affine_solver為第3.1節所提到的仿射映射(affine mapping)寫成的程式碼，輸出結果是用QR分解的方法找出旋轉以及平移的矩陣

```
def sort_idx(value,method = 'ascend'):
  x = sorted(enumerate(value),key = operator.itemgetter(1))
  m = len(value)
  temp = zip(*x)
  sorted_value = temp[1]
  idx = temp[0]
  if method == 'ascend':
    return np.array(sorted_value),np.array(idx)
  elif method == 'descend':
    sorted_value = list(sorted_value)
    sorted_value.reverse()
    idx = list(idx)
    idx.reverse()
    return np.array(sorted_value),np.array(idx)
  else:
    print 'The type of method must be ''ascend'' or ''descend''.'
```

```
    return
```

可依照'ascend'以及'descend'排序出遞增或遞減的值，而輸出結果是排序後的值以及排序前的位置

```
def pdist(A):
  """
  The pair-wise distance of matrix A, (m,n) = A.shape
  A is the row data of n dimensions with m samples
  Syntax: D = pdist(A)
  For example:
  ==============
  A = np.random.random([8,2])
  D = pdist(A)
  """
  start = time.time()
  m,n = A.shape
  D = list()
  for i in range(m):
    for j in range(m-i-1):
      D.append(la.norm(A[i]-A[i+j+1]))
  AT = time.time() - start
  print "Pdist Running Time: %s" % AT
  return np.array(D)
```

輸出結果D為A自己的列與列之間的距離，再搭配下述所提供的squareform即可造出距離矩陣

```
def squareform(D):
  """
  Reshape the vector type pair-wise distance vector to the square form.
  Syntax: D = squareform(D)
  """
  if len(D.shape)<>1:
```

```
    print 'D must be one dimensional vector'
    return
  #check the length of D is available
  n = len(D)
  m = 1
  while ((m-1)*m)/2 < n:
    m+=1
  if ((m-1)*m)/2 <> n:
    print 'D does not come from pdist function'
    return
  #initial a zero matrix
  A = np.zeros([m,m])
  k = 0
  for i in range(0,m-1):
    for j in range(i+1,m):
      A[i,j] = D[k]
      k+=1
  return A+A.T
# 輸出結果是把pdist的結果造出距離矩陣

def D2X(D,n_eig):
  """
  D is distance matrix
  """
  m,n = D.shape
  if m == n:
    H = np.eye(n)-np.ones(n)/n
    M = -np.dot(np.dot(H,D**2),H)/2
    DM,V = la.eig(M)
    DM = np.real(DM)
    Y,ID = sort_idx(DM,'descend')
    ID = ID[:n_eig]
```

```
    if n_eig == len(Y):
      err = 0
    else:
      err = abs(Y[n_eig])
    V = V[:,ID]
    DM = DM[ID]
    X = np.real(np.dot(cnp.sqrt(np.diag(DM)),V.T))
    return X,err
  else:
    print 'D must be a distance matrix'
```

輸出結果就是把資料D(距離矩陣)透過MDS計算出以n_eig為維度的矩陣X

```
def kron(A,m,n):
  """
  kron a big matirx from A to m-times in rows and n-times in column.
  Syntax: B = kron(A,m,n)
  If A is a vector, kron consider A to a row vector.
  """
  if len(A.shape)==1:
    A = np.array([A])
  m1,n1 = A.shape
  B = np.zeros((m1*m,n1*n))
  for i in range(m*m1):
    for j in range(n*n1):
      B[i,j] = A[i/m,j/n]
  '''
  for i in range(m):
    for j in range(n):
      B[i*m1:(i+1)*m1,j*n1:(j+1)*n1] = A
  '''
  return B
```

在此的kron是把矩陣A作擴充，把矩陣A的列擴充m倍、行擴充n倍

```
def scmdscale(D,p,Ng,Ni,r):
```

D是距離矩陣，p是欲降至的維度，Ng是子資料寬度，Ni是重疊的子資料中交集的寬度，r是讓拆解的部分可以隨機選取

```
  start = time.time()
  n,m = D.shape
  N = n
  if r == 0:
    rand_id = np.array(range(N))
  else:
    rand_id = np.random.permutation(N)
```

這裡的r可以控制是否隨機選取

```
  Nout = p
  K = np.floor((1.0*N-Ng)/(Ng-Ni))+1
  X = np.zeros((Nout,N))
```

先宣告X為Nout*N的矩陣，接著把拆解合成後的資料一個個貼上來

```
  n1 = 1
  n2 = Ng + np.mod((N-Ng),(Ng-Ni))
  k = 1
  while n2 <= N:
    D1 = D[rand_id[n1-1:n2]].T[rand_id[n1-1:n2]].T
```

D1為每次拆解的部分

```
    X1 = D2X(D1, Nout)
    X1 = X1[0]
```

X1為拆解的部分作MDS降至Nout的維度

```
    if k ==1:
      X[:,n1-1:n2] = X1
    else:
      U,b = affine_solver(X1[:,0:Ni].T,X[:,n1-1:n1+Ni-1].T)
```

在此的U,b則是透過affine_solver所算出的旋轉以及平移矩陣

```
      X1 = np.dot(U,X1) + kron(b,Ng,1).T
```

```
217|       X[:,n1-1:n2] = X1
        # X1透過旋轉平移，重疊部分的值已經和X中重疊的值一模一樣了，所以值可以直接合成
218|     n1 = n2-Ni+1
219|     n2 = n2+Ng-Ni
220|     k = k+1
221|   temp = np.zeros((Nout,N))
222|   temp[:,rand_id] = X.copy()
223|   X = temp.copy()
224|   X = X.T
225|   X = zero_sum(X)
226|   M = np.dot(X.T,X)
227|   L,basis = la.eig(M)
228|   Z,ID = sort_idx(L)
229|   basis = basis[:,ID]
230|   L = L[ID]
231|   Y = np.dot(X,basis)
232|   m1,n1 = Y.shape
233|   temp = np.zeros((m1,n1))
234|   for i in range(n1):
235|     temp[:,i] = Y[:,-i-1]
236|   Y = temp
237|   print "Scmdscale Running Time: %s" % (time.time() - start)
238|   return Y
    # Y即為SCMDS所算出的結果

239| class shared_array(object):
240|   """
241|   Allocate a shared memory as array type.
242|   Syntax: A = shared_array((m,n)) #create a m-by-n shared array object
243|   For example:
244|   =============
```

```
  A = shared_array((3,4))
  A.array  #show the detail of shared array A
  A.array.shape  #show the matrix size
  A.array[i,j] = k #indicate the (i,j) element of A as k
  """
  def __init__(self,matrix_shape=(1,1)):
    (m,n) = matrix_shape
    self.shared_base = mp.Array(ctypes.c_double,m*n)
    self.array = np.ctypeslib.as_array(self.shared_base.get_obj())
    self.array = self.array.reshape(m,n)
    self.shape = self.array.shape
# 這裡的shared_array與2.2所提到的shared_array是一樣的

def array2shared(A,shared_A):
  m1,n1 = A.shape
  m2,n2 = shared_A.shape
  if m1<>m2 or n1<>n2:
    print 'The size of matrices must be the same'
    return
  else:
    for i in range(m1):
      shared_A.array[i] = A[i]
# 這裡的array2shared與2.2所提到的array2shared是一樣的
```

附錄B：SC-MDS拆解平行化版本的code

SC-MDS拆解平行化版本的code

```
265| def pscmdscale(D,p,Ng,Ni,r):
   # D是距離矩陣，p是欲降至的維度，Ng是子資料寬度，Ni是重疊的子資料中交集的寬度，r是讓拆解的部分可以隨機選取
266|   start = time.time()
267|   n,m = D.shape
268|   def scmds(shared_D,idx,rand_id,Nout,shared_X,k):
269|     D1 = shared_D.array[rand_id[idx[k][0]-1:idx[k][1]]].T[rand_id[idx[k][0]-1:idx[k][1]]].T
270|     X1 = D2X(D1, Nout)
271|     X1 = X1[0]
272|     if k ==0:
273|       shared_X.array[:,0:idx[0][1]] = X1.copy()
274|     else:
275|       shared_X.array[:,idx[0][1]+(k-1)*Ng:idx[0][1]+k*Ng] = X1.copy()
     # 在此的scmds這個副函式是用來讓每個pid可以同時間個別執行MDS，並把結果放入shared_X這個共享矩陣中
276|   N = n
277|   if r == 0:
278|     rand_id = np.array(range(N))
279|   else:
280|     rand_id = np.random.permutation(N)
     # 這裡的r可以控制是否隨機選取
281|   Nout = p
282|   K = np.floor((1.0*N-Ng)/(Ng-Ni))+1
283|   n1 = 1
284|   n2 = Ng + np.mod((N-Ng),(Ng-Ni))
285|   idx = list()
286|   while n2<=N:
```

```
287|     idx.append((n1,n2))
288|     n1 = n2-Ni+1
289|     n2 = n2+Ng-Ni
290|   X = np.zeros((Nout,int((K-1)*Ng+idx[0][1]-idx[0][0]+1)))
     # 先宣告X為Nout*int((K-1)*Ng+idx[0][1]-idx[0][0]+1)的矩陣，
用來把拆解平行化的資料可以同時貼上來，之後會再進行單核心的合併
291|   if __name__=='scmds':
292|     #allocated shared memory
293|     shared_X = shared_array((Nout,int((K-1)*Ng+idx[0][1]-idx[0][0]+1)))
294|     #make shared_X function
295|     assert shared_X.array.base.base is shared_X.shared_base.get_obj()
296|     #copy data from matrix to shared array
297|     array2shared(X,shared_X)
298|     mm,nn = D.shape
299|     #allocated shared memory
300|     shared_D = shared_array((mm,nn))
301|     #make shared_D function
302|     assert shared_D.array.base.base is shared_D.shared_base.get_obj()
303|     #copy data from matrix to shared array
304|     array2shared(D,shared_D)
305|     procs = [mp.Process(target=scmds, args=
(shared_D,idx,rand_id,Nout,shared_X,k)) for k in range(len(idx))]
       # 這裡要把平行運算的副函式scmds放入procs這個list裡
306|     for p in procs: p.start()
307|     for p in procs: p.join()
       # 此為執行proc整個list，整個list執行完則代表個別執行完副函
式scmds並且放入shared_X等待合成
308|     for k in range(len(idx)-1):
309|       U,b = affine_solver(shared_X.array
[:,idx[0][1]+k*Ng:idx[0][1]+k*Ng+Ni].T,
shared_X.array[:,idx[k][1]-Ni:idx[k][1]].T)
310|       X1 = np.dot(U,shared_X.array
```

```
[:,idx[0][1]+k*Ng:idx[0][1]+(k+1)*Ng]) + kron(b,Ng,1).T
        shared_X.array[:,idx[k+1][0]-1:idx[k+1][1]] = X1.copy()
    # 上述的for迴圈是把合成的部分用單核心的方法在shared_X中接上
    temp = np.zeros((Nout,N))
    temp[:,rand_id] = shared_X.array[:,0:N].copy()
    X = temp.copy()
    X = X.T
    X = zero_sum(X)
    M = np.dot(X.T,X)
    L,basis = la.eig(M)
    Z,ID = sort_idx(L)
    basis = basis[:,ID]
    L = L[ID]
    Y = np.dot(X,basis)
    m1,n1 = Y.shape
    temp = np.zeros((m1,n1))
    for i in range(n1):
        temp[:,i] = Y[:,-i-1]
    Y = temp
    print "PScmdscale Running Time: %s" % (time.time() - start)
    return Y
    # Y即為拆解平行化的SCMDS所算出的結果
```