

中 華 大 學
碩 士 論 文

預測修正法使用 Python 在分散式計算環境
問題的探討

Block predictor – corrector method on
Python-based distributed computing
environment

系 所 別：應用統計學系碩士班

學號姓名：M09109015 鄭至傑

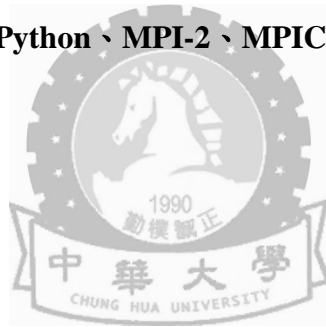
指導教授：李明恭 博士

中 華 民 國 100 年 8 月

摘要

本論文在探討使用 Python 對區域預測-修正法(Block Predictor -Corrector method)進行分散式平行運算，在平行化過程中都使用 MPI-2 (Message Passing Interface)做為各個 CPU 互相溝通的基本標準，應用程式 MPICH2(Message Passing Interface Chameleon)來讓每個 CPU 在執行程式時方便與其它 CPU 進行資料的傳送、接收，其控制與操作則是由 Python 所安裝的套件 mpi4py 使用各種函數 send、recv、get_size、get_rank 等進行，在計算區域預測修正法平行方式的探討與研究方面，則是以一台個人電腦(Personal Compute, PC)與兩台個人電腦(Personal Compute, PC)以同樣例題做比較。

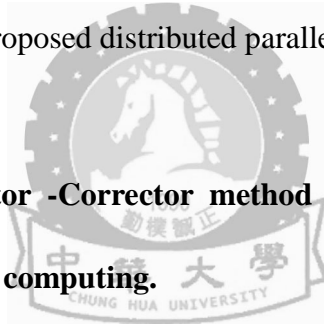
關鍵詞：區域預測-修正法, Python、MPI-2、MPICH2、mpi4py、分散式平行運算。



英文摘要

In this thesis, we study how to apply Python to do parallel computation in a distributed computing structure. MPI-2 (Message Passing Interface) is implemented to be a basic standard on parallel processing to communicate between each CPU inside the structure. MPICH2 (Message Passing Interface Chameleon) is implemented to let individual CPU to do data transmission and reception between other CPUs. While controlling and manipulation of the process is proceed by applying the functions; send 、recv 、get_size 、get_rank, etc, in the package of mpi4py by Python. Numerical experiments are carried on the block method to solve some stiff ordinary differential equations, and their speed up between 1 CPU and two CPU are explored to show positive achievement of the proposed distributed parallel structure.

Keyword: Block Predictor -Corrector method , Python, MPI-2, MPICH2, mpi4py, distributed parallel computing.



致謝

本論文的完成首先感謝恩師 李明恭教授在中華大學求學的這些日子裡，總是不辭辛勞的指導我，不論是課業上還是做人處事方面，都是盡全力的栽培我，在此還要感謝 羅琪副教授也在口試之前給我許多指導與建議，在這裡致上由衷敬意。

在學期間，非常感謝指導我論文的學長 睿緯，以及我的研究室的學弟與同學 文祥、炯良、嵐迪、宇評、欣全、昭傑、效豪與我在實驗室研究、討論甚至還將做論文的心得分享給我，並且互相勉勵，在此還有要感謝我的女朋友 文慧以及一直支持我的父母、家人在我做論文期間不斷給我支持與包容，與不厭其煩的叮嚀。

最後再次感謝我身邊所有的人，因為有他們我才能順利完成學業，我將這分成果與喜悅分享給他們。



目錄

摘要.....	i
英文摘要.....	ii
致謝.....	iii
目錄.....	iv
圖目錄.....	v
第一章 背景與目的.....	1
第二章 數值方法.....	3
2.1 Predictor-Corrector method	4
2.2 Block Predictor-Corrector method.....	5
第三章 平行處理.....	7
3.1 平行系統.....	8
3.3 MPI – 2	9
3.4 MPICH2.....	11
3.5 Python	11
3.6 MPI for Python	12
第四章 平行運算進行PECE of Block Method	16
4.1 環境建立.....	16
4.2 方法.....	17
4.3 比較.....	25
第五章 結論與展望.....	27
第六章 參考文獻.....	28
第七章 附錄.....	29
附錄一(建立平行環境的安裝).....	29
附錄二(未平行的區域預測修正法的主程式).....	33
附錄三(平行的區域預測修正法的主程式).....	42
附錄四(程式中運算的資料類型與C語言的對照)	47

圖目錄

圖-1 平行處理概念圖	7
圖-2 SMP圖	8
圖-3 MPP圖	9
圖-4 MPI.....	10
圖-5 PECE-在單CPU上	23
圖-6 PECE-在兩個CPU上	24
圖-7 wmpiregister.....	29
圖-8 wmpiconfig	30
圖-9 hosts data saved successfully	31
圖-10 import mpi4py	31
圖-11 helloworld.....	32



第一章 背景與目的

對於現在越來越發達的科技，由於需要電腦處理的計算更加的複雜，運算時讓結果更為準確、錯誤更少，並且還需要能同時處理這麼多的繁雜的運算式子達到同步、即時、快速的效果，為了達到這些期望與需求，我們必須要有好的方法還要配上好的配備，使用好的方法能減少不必要的運算，確實降低硬體資源的消耗與浪費，好的配備能強化各種運算的速度，並且能比低階配備更具有同時處理多項工作的能力。

在做一個複雜的求解運算時，整個計算過程中因為疊代次數繁多或是矩陣龐大，導致一般單一臺個人電腦進行運算時的負擔過重，而且會影響運算產生的結果輕則效率上受到些微延遲，重則會因運算的程式當掉無法進行接下來的運算，為了避免出現效率降低或是無法繼續執行運算的情況發生，而使用電腦叢集進行分工來處理龐大電腦記憶體需求的問題，並且大大提升原本運算上的效率，一般矩陣基本計算大多採用直接法，但是在面對較大型的結構分析上因為龐大的電腦記憶體需求導致較為困難且耗時，故可採用多台電腦平行處理運算來提升時間上的效率。

要能準確將所要的結果運用在之後預測函數結果的計算上面，可採用區域預測修正法估計函數值，首先將現有的已知解帶入預測方程式取得預測值，再利用修正方程式將其預測值帶入並修正取得修正後的預測值，修正後的預測值經過數次修正之後其誤差到達可接受的範圍之後將此結果視為解，並將此解做為函數下一個已知解並繼續進行估計與修正完成之後函數估計及求解過程。

本論文使用 Python 內的 mpi4py 撰寫運算程式並搭配以 MPI 為基礎的 MPICH2 平行介面應用程式對各電腦進行平型的控制，再利用電腦叢集的整合對區域預測 - 修正法(Block predictor – corrector method)進行運算來創造出一個平行計算處理的環境，然後利用平行處理進行預測 - 修正法裡面的計算，其運作方式是讓巨大的矩陣分工給許多效能較低的電腦處理，並且把矩陣內的複雜

計算分散給多個 CPU 處理，進而求解。



第二章 數值方法

差分方程是一個很重要的數學工具，它可以解決許多領域的問題，在一般情況我們特別注意系統所生成的數值解的初始值問題(IVP)。我們首先考慮最簡單的初值問題在常微分方程(ODEs)的一般形式如下：

$$y'(t) = f(t, y)$$

$$y(t_0) = y_0$$

$$t, t_0 \in [a, b]$$

a, b 是已知。

利用數值方法求解時，無法求出以函數表示之解析解，而是找出在一步一步的節點上其近似解。在大部份的情況下，各個節點的間距 h 通常都是定值，數值方法就是要找出各節點上 y 的近似解，一階尤拉法是求解初始值常微分方程式最簡單的方法，其計算公式如下

一階尤拉法(the first order Euler method)

$$y_{n+1} = y_n + hf(x_n, y_n)$$

除了尤拉法還有

四階 龍格庫塔(the fourth order Runge-Kutta method , RK4)

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_2\right)$$

$$k_4 = f(t_{n+1}, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h, n \geq 0$$

龍格庫塔法求的 $i+1$ 數值解只需用到 i 點的資料，與 i 點之前的資料無關，像 龍格庫塔法這樣，只用前一點的資訊就可求出推算求下一點的解，稱為單步法(single-step method)。初始條件為已知，如果使用單步法，就可以由起始點，一



點一點的求出其後各點的解答。

但是利用這類方法求得部分資料以後，我們對所積分的函數上已經有了額外的資料，此時如果能夠善用計算機的記憶，利用需要較多資料的多步法(multi-step method)，是可以讓積分準確性及穩定性都進一步提昇，例如亞當斯顯式法(Adams-Bashforth method)

$$y_{n+1} = y_n + \frac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}), n \geq 3$$

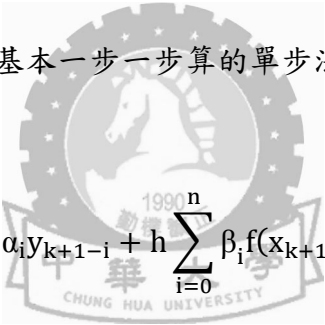
和

四階亞當斯隱式法(fourth order Adams-Moulton method)

$$y_{n+1} = y_n + \frac{h}{24}(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}), n \geq 2$$

2.1 Predictor-Corrector method

在常微分方程計算中從基本一步一步算的單步法演變到多步法，其中利用下面多步法計算公式：


$$y_{k+1} = \sum_{i=1}^n \alpha_i y_{k+1-i} + h \sum_{i=0}^n \beta_i f(x_{k+1-i}, y_{k+1-i})$$

參數 α_i 和 β_i 由多項式差值求出。上面公式則是由前面幾個節點的資訊($f(x_i, y_i), i = k+1-n, \dots, k$)，求出一個差分多項式，將這個多項式外延(extrapolation)，並且將其積分遍可得到上式。當 $\beta_0 = 0$ 則此這計算式為顯式，反之， $\beta_0 \neq 0$ 則這計算式為隱式，隱式法通常比顯式法準確也比顯式法穩定，但隱式法須有 y_{k+1} 的初始預測值，初始預測值可由顯式法求得，因此構成所謂的預測 - 修正法(predictor - corrector)，預測 - 修正法中先用顯式法求出預測值，然後再用隱式法進行修正。例如最廣泛使用的多步法，利用公式亞當斯顯示法作為一個預測公式部分，四階亞當斯隱式法作為一個修正公式的部分，產生一個四階亞當斯預測 - 修正法(fourth order Adams type predictor - corrector method ABM)

$$y_{n+1}^p = y_n^c + \frac{h}{24} (55f_n^c - 59f_{n-1}^c + 37f_{n-2}^c - 9f_{n-3}^c)$$

$$y_{n+1}^c = y_n^c + \frac{h}{24} (9f_{n+1}^p + 19f_n^c - 5f_{n-1}^c + f_{n-2}^c)$$

式子中 y 和 f 的 p 和 c 分別表示預測與修正值。

上式中產生的預測修正 EP(EC) 模式的概念可以寫成以下流程：

$$L \rightarrow F_n^p \rightarrow y_n^p \rightarrow F_n^c \rightarrow y_n^c \rightarrow L$$

$$F_n^p = f\left(\begin{matrix} x_{n-1} \\ x_n \end{matrix}, y_{n-1}^c\right) \quad \text{和} \quad F_n^c = f\left(\begin{matrix} x_n \\ x_{n+1} \end{matrix}, y_n^p\right)$$

雖然修正值可以重複疊代，直到符合收斂條件為止，但一般並不值得如此計算，

通常修正值只做一次，此種方法稱為 PECE 法。其步驟為

P (predictor)

E (evaluation)

C (corrector)

E (evaluation)

多步法雖然比單步法穩定和準確，但多步法沒有辦法靠自己自行啟動 (self-starting)，通常須先用單步法求出開始幾點，才能代入多步法公式，因此寫程式較為困難。

2.2 Block Predictor-Corrector method

區域數值解法使用在剛性與非剛性的常微分方程已經許久，在各種區域法的演化也各有不同，本論文主要使用的是以類似於牛頓法(Newton-like methods)的區域數值解法，在此多階(S)和多步(M)方法求解的常微分方程由於它擁有明確的隱式所以可以進行預測修正，此方法由學者Ming-Gong Lee [3]中提出並討論，對於 $S=2$ ， $M=3$ 的區域預測修正法顯性的公式如下：

$$\begin{aligned} 2y_{n+2} - 9y_{n+1} + 18y_n - 11y_{n-1} &= 6hf(x_{n-1}, y_{n-1}) \\ -y_{n+2} + 6y_{n+1} - 3y_n - 2y_{n-1} &= 6hf(x_n, y_n) \end{aligned}$$

$$n \geq 1$$

經過整理改寫之後：

$$\begin{bmatrix} y_{n+1}^p \\ y_{n+2}^p \end{bmatrix} = \begin{bmatrix} 5 & -4 \\ 28 & -27 \end{bmatrix} \begin{bmatrix} y_{n-1} \\ y_n \end{bmatrix} + \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 6h \cdot f(x_{n-1}, y_{n-1}) \\ 6h \cdot f(x_n, y_n) \end{bmatrix}$$

$n \geq 1$

其隱式如下：

$$\begin{aligned} 2y_{n+2} + 3y_{n+1} - 6y_n + y_{n-1} &= 6hf(x_{n+1}, y_{n+1}) \\ 11y_{n+2} - 18y_{n+1} + 9y_n - 2y_{n-1} &= 6hf(x_{n+2}, y_{n+2}) \end{aligned}$$

$n \geq 1$

經過整理改寫之後：

$$\begin{bmatrix} y_{n+1}^c \\ y_{n+2}^c \end{bmatrix} = \begin{bmatrix} \frac{5}{23} & \frac{28}{23} \\ -\frac{4}{23} & \frac{27}{23} \end{bmatrix} \begin{bmatrix} y_{n-1} \\ y_n \end{bmatrix} + \begin{bmatrix} \frac{11}{69} & \frac{-2}{69} \\ \frac{6}{23} & \frac{1}{23} \end{bmatrix} \begin{bmatrix} 6h \cdot f(x_{n+1}, y_{n+1}^p) \\ 6h \cdot f(x_{n+2}, y_{n+2}^p) \end{bmatrix}$$

$n \geq 1$

之後的章節將利用此區域預測 - 修正法公式進行平行電腦的運算。



第三章 平行處理

在現代個人電腦的運算處理速度已經相當的快速，不過使用者總是希望速度可以越來越快，處理的工作可以越做越多，可是卻希望消耗的成本不會隨著速度越來越快與工作量越來越大而跟著提高；將電腦叢集的組合進行平行式的運算，將較為慢的、舊的電腦串連起來，利用網路連結將資料互相傳遞並進行運算，把一個程式分割為若干個任務，在將每一個任務分送到不同的 CPU 上面進行運算，之後再將運算完的資料整合進而得到結果，成本上比新置一台較為高速的電腦節省許多，效率上可以達到接近購置高速電腦的速度甚至超過其速度，要將各台電腦串連起來互相溝通，就要有讓各臺電腦能進行溝通的管道，本論文所使用的溝通管道的應用程式是 MPICH2，利用 Python 程式語言使用 mpi4py 函式庫對 MPICH2 進行控制。

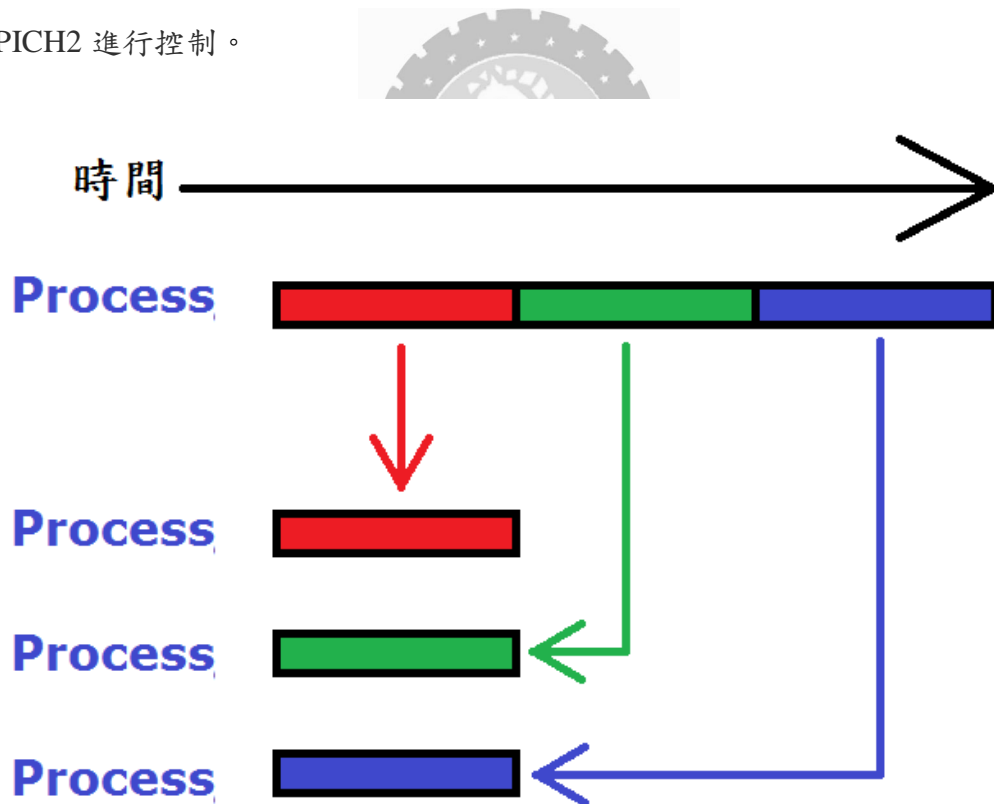


圖-1 平行處理概念圖

3.1 平行系統

平行運算的主要架構有兩種，共用式記憶體系統(Shared Memory Parallel SMP System)還有分散式記憶體系統(Massive Parallel Processors MPP System)兩種，共用式記憶體系統由多個同型 CPU 共用一組記憶體，這樣的系統就是平行電腦(Parallel Computer)一台電腦擁有多個 CPU，並且是利用十字交叉網(Crossbar)或是系統匯流排(System - bus)進行連結這些 CPU。如圖

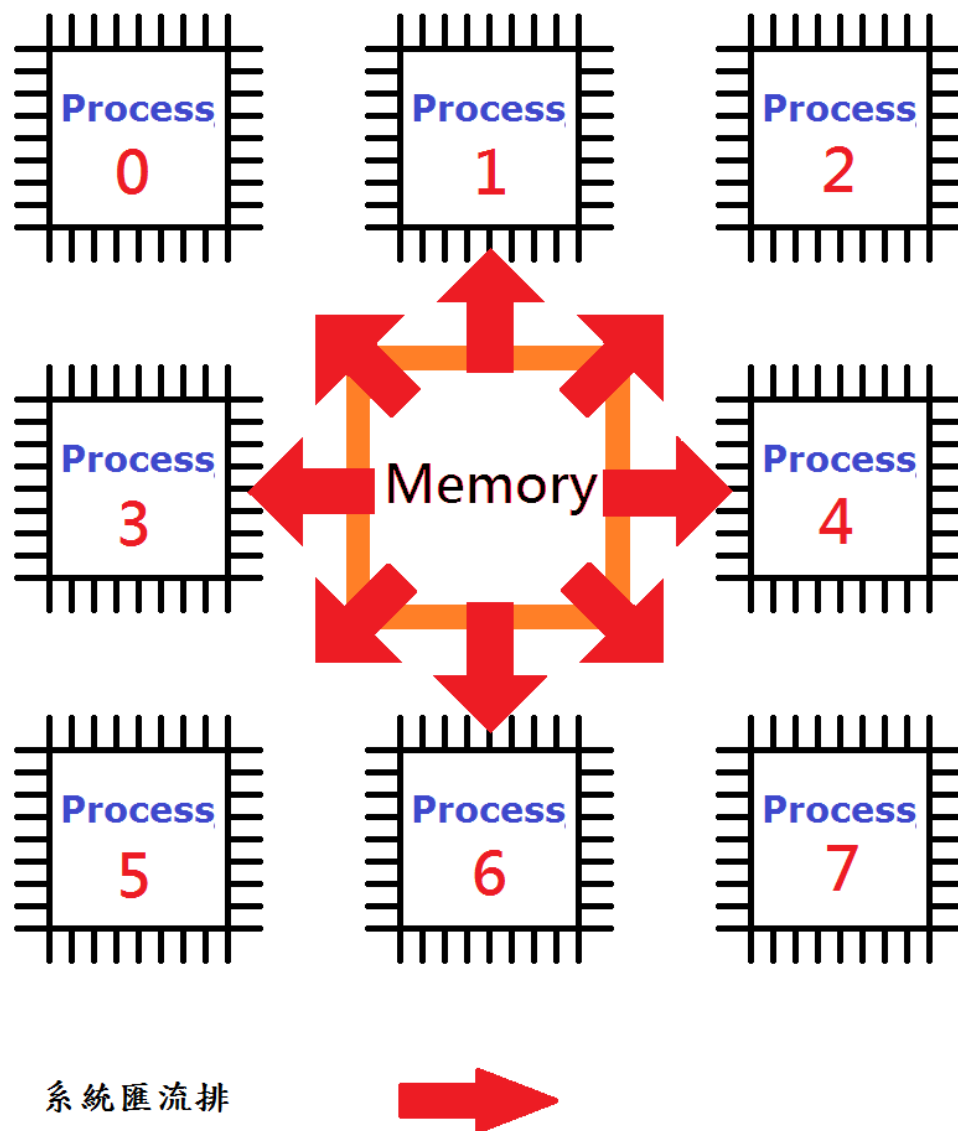


圖-2 SMP 圖

分散式記憶體系統則是不同於共用式記憶體系統，每個 CPU 各自擁有自己獨立記憶體、操作系統，此系統通常是由多個單獨 CPU 的電腦組成的電腦叢集 (Computer Clusters)，電腦 CPU 與另外一台電腦的 CPU 之間則是由乙太網路所連結 (Ethernet)，兩種系統各有優缺點，共用記憶體系統擁有方便控制與操作，讓使用者甚至感覺不出來在操作各 CPU 進行平行的資料處理，不過在硬體擴充方面可以增加 CPU 的空間卻十分有限，相反的分散式記憶體系統是由電腦叢集組成，在擴充方面不論是記憶體或是 CPU 都非常容易，不過在控制上就要利用程式進行控制、工作分配相對於共用式記憶體系統來的繁雜許多。

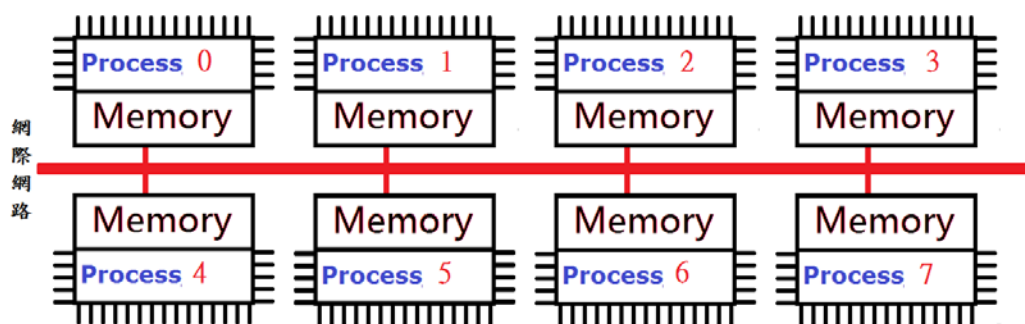


圖-3 MPP 圖

如圖(3)本論文主要使用 MPP 平行系統的方式將電腦與電腦之間彼此互相連接再利用 MPICH2 平行應用程式將連接起來的電腦可以互相傳遞訊息。而 SMP 的平行電腦則是當作比較與對照之用。

3.3MPI – 2

MPI(Message Passing Interface)代表訊息傳遞的界面，MPI是一個的訊息傳遞界面的標準，於 1993 年 11 月在 93 超級計算機會議時MPI標準的草案被提交，有超過 115 個單獨函數定義在其中，最終版本的草案發布於 1994 年 5 月且從MPI進升到MPI - 2，不過MPI - 2 還是依照著 MPI的 規範，並發表專題強化MPI的規範，原來的MPI在後來被稱為的MPI - 1。

MPI - 2 幾乎支持所有高性能計算平台(包括 Linux 和 Mac OS x)還有 Windows, 並且能很容易將兩台或是兩台以上的電腦之間進行資料的傳輸、交換, 取代了之前的所有消息傳遞庫, 就像是一個萬用的水管分接頭能將各種不同平台的水管接起來, 並且可以加以控制各個水管中的流向該給哪個 CPU 的資料就會準確的將資料傳給指定目標, 也可以將已經傳給目標經過處理計算之後的結果傳回發訊主機進行整合, MPI 內含上百個函數讓使用者可以很有效率的對目標進行溝通。

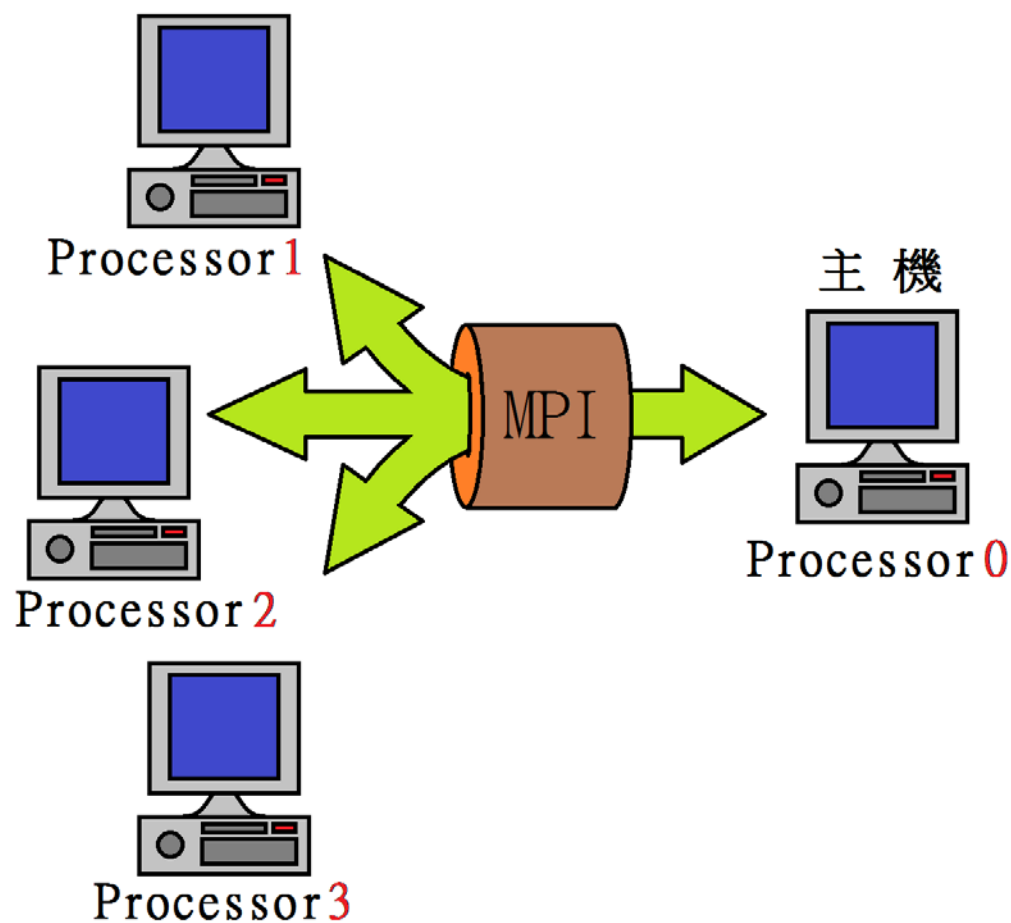


圖-4 MPI

以下為 MPI 標準規範：

點對點傳訊 (Point to Point Comm.)

CPU 對另外一個 CPU 的溝通。

集體傳訊 (Collective Comm.)

一個 CPU 對多個 CPU 的溝通或是多個 CPU 對一個 CPU 的溝通。

行程組 (Process Groups)

將各 CPU 分組以便於集體傳訊的效率。

傳訊情況 (Contexts)

利用對於各 CPU 溝通狀況的掌控。

行程拓撲 (Process Topologies)

對訊息溝通的編成與規劃。

Fortran 和 C 的整合 (Bindings)

對於 C 和 Fortran 的整合使溝通更為方便簡單。

3.4 MPICH2

MPICH2 中的 MPI 就是代表訊息傳遞的界面 MPI(Message Passing Interface)，而 CH(Chameleon)則是取自於變色龍這單字，它由美國 Mathematics and Computer Science Division 的 Argonne 實驗室所發展，MPICH 是以 C 語言實作使用 MPI 標準的應用軟體，內容包括 MPI- 1 和 MPI- 2 的全部功能實際應用。

MPICH2 是一種開放源碼應用程式被廣泛用於高性能計算集群，MPICH2 內含最基本函數 MPI_COMM.send、MPI_COMM.recv、MPI_COMM.rank、MPI_COMM.size，還包含其它 MPI 裡的設定與函數控制的實際操作做函數，讓平行化便利且容易執行操作，執行的程式不用修改原代碼只要決定所用到的 MPICH 函數並給予所需的參數就可連結到一個有安裝 MPICH2 的不同平台上進行溝通、控制，不只如此，在多核心架構裡 MPICH2 它會不需要特別設定，它是一個擁有自動判斷識別的功能，還會將通訊品質提升。

3.5 Python

Python 是一種直譯式高階電腦語言，他不僅有完整的物件導向特性，而且可以在 Microsoft Windows、Linux，以及 Macintosh 作業系統下執行。Python 的程

式碼簡潔，而且提供大量的程式模組，這些程式模組可以幫助我們快速撰寫程式，比起 C 或是 Java 來，Python 往往只需要數行程式碼，就可以做到其他語言數十行程式碼的工作，不管你是初學者還是已經有數年經驗的專家，都可以快速的學會 Python，並且建立高效率的 Python 程式碼，Python 是一種直譯式的電腦語言語法簡潔易讀而且直譯器是使用 C 所寫成的，程式模組大部分也是使用 C 寫成而且程式碼是公開更能方便我們研究，而且 Python 的程式碼是公開的，全世界有無數人在搜尋 Python 的 bugs，並且修訂他。而且不斷地新增功能，此外 Python 提供內建直譯器，我們可以在直譯器內建立、測試，與執行程式碼，而不需要額外的編譯器，也不需要經過編譯的步驟，我們不需要完整的程式模組才能測試，只需要再直譯器下所需測試的部分就可以，Python 直譯器非常的有彈性，它允許遷入 C++ 的程式碼，來做延伸模組。

Python 除了可以在多種類作業系統中直行之外，不同種類的作業系統使用的程式介面也是一樣。我們可以在 Macintosh 上撰寫 Python 程式碼，在 Linux 上測試，然後上載到 Windows 上執行，都是使用相同程式碼。當然這是對大部份的 Python 模組而言，有少部分的 Python 模組還是針對特殊的作業系統而設計的。

3.6 MPI for Python

由於 MPICH 是以 C 語言實作 MPI 定義的應用軟體，所以 Python 本身並未有 MPI 的溝通模組，而 mpi4py，則是 Python 上使用的平行化程式套件，底層仍需要 MPICH 套件來做 node 間 process 的傳遞，但在開發上能使用 Python 簡潔美觀的語法。

使用 MPI4PY 讓 Python 可以套用 MPICH2 的函式庫對各個電腦進行平行化的運算控制，MPI4PY 的基本與法：

MPI.COMM_WORLD.Get_name

MPI.COMM_WORLD.Get_size

MPI.COMM_WORLD.Get_rank

通常在呼叫 MPI.COMM_WORLD.Get_size 以得知此訊息傳遞系統參與平行計算的 CPU 總數，及呼叫 MPI.COMM_WORLD.Get_rank 以得知當前使用的是第幾個 CPU，在描述第幾個 CPU 是從 0 開始起算，所以第一個 CPU 的 Get_rank 值為 0，第二個 CPU 的 Get_rank 值為 1，第三個 CPU 的 Get_rank 值為 2，其餘依此類推。

通常要在幾個 CPU 上做平行計算是在下執行命令時決定的，而不是在程式裏事先設定。當然，使用者也可以在程式裡事先設定要在幾個 CPU 上作平行計算，實際上使用幾個 CPU 作平行計算是根據 -n 或 -np 的設定值，在設定 CPU 的 size 上面可以超過 CPU 總數，不過在執行時多的 rank 將會被分配到已執行過的 CPU 式上面，也就是說可能 Get_rank 值 0 跟 3 都是被同一個 CPU 執行的，為了讓顯示 CPU 的執行狀況更明卻並且更能有效控制 CPU 工作的分配則可利用 MPI.COMM_WORLD.Get_name，取得該 CPU 所屬於的電腦名稱。

Get_name、Get_size、Get_rank 的呼叫格式如下：

MPI.COMM_WORLD.Get_name(self)

MPI.COMM_WORLD.Get_rank(self)

MPI.COMM_WORLD.Get_size(self)

引數 MPI_COMM_WORLD 是 MPI 內定，在參與該程式平行計算的全部 CPU 都是屬於同一個訊息傳遞系統(communicator)。屬於同一個訊息傳遞系統的各個 CPU 之間才可以傳送資料控制。

MPI.COMM_WORLD.send

MPI.COMM_WORLD.recv

參與平行計算的各個 CPU 之間的資料傳送方式有兩種，一種叫做點對點通訊 (point to point communication)，另一種叫做集體通訊(collective communication)。一個 CPU 與另一個 CPU 之間的傳送屬於點對點通訊，MPI.COMM_WORLD.send 和 MPI.COMM_WORLD.recv 就是屬於點對點通訊方

式的傳輸函數。送出資料的 CPU 要呼叫 MPI.COMM_WORLD.send 來送資料，而收受資料的 CPU 要呼叫 MPI.COMM_WORLD.recv 來收資料。一個 send 必須要有一個對應的 recv 與之配合，才能完成一份資料的傳送工作。

send 的呼叫格式如下：

```
MPI.COMM_WORLD.send(self, obj=None, int dest=0, int tag=0)
```

參數

obj 要送出去的物件起點，可以是純量(scalar)或陣列(array)資料。

int dest 是接收資料的 CPU 的編號。

int tag 將此傳送的過程加上標籤與 recv 函數對應，防止傳訊息中連續傳送時使資料發送到同樣來源的 CPU 卻是要接收另外一個發送點的資料。

recv 的呼叫格式如下：

```
MPI.COMM_WORLD.recv(self, obj=None, int source=0, int tag=0, Status status=None)
```

參數

obj 要送出去的物件起點，可以是純量(scalar)或陣列(array)資料。

int source 是發送資料的 CPU 的編號。

int tag 將此傳送的過程加上標籤與 send 函數對應，防止傳訊息中連續傳送時使資料接收到同樣來源的 cpu 卻是要傳給另外一個接收點的資料。

Status status 是執行 recv 副程式之後的狀況。

在傳送資料（ send, recv ）時，是以下列四項構成其信封(envelope)，用以識別一件訊息(message)。

1. 送出資料的 CPU id
2. 收受資料的 CPU id
3. 資料標籤
4. 訊息傳遞系統

所以一個 CPU 送給另外一個 CPU 多種資料時，不同的資料要用不同的資料標籤，以資識別。



第四章 平行運算進行 PECE of Block Method

本論文利用平行環境對 $S = 2, M = 3$ 的區域預測修正法進行例題運算，利用 Python 程式對電腦 CPU 的控制，將預測與修正法可同時進行的部分分給不同 CPU 運算，分開運算後的結果在送到 CPU0 進行整合算出解。

4.1 環境建立

在要使用平行運算之前將平行的環境建立起來所需要用到的各種硬體、軟體也全部安裝，硬體方面配備使用的是第一種 Acer Veriton M461 與第二種 Acer Veriton 6800(兩台)來進行區域網路的連線，兩台電腦的規格如下：

第一種

Product/Model: Veriton 6800

CPU: Intel Pentium 4 3.20 GHz

RAM: 512 MB

BIOS: Phoenix Award BIOS v6.00PG R01-A4

Host Bus Adapter : Intel Integrated 82801GB ICH7 , Serial ATA Controller

DVD : LITE-ON DVD SOHD-16P9S 16X , ATAPI

Hard Disk: Seagate Barracuda ST3160023AS , Serial ATA 160G

Video Adapter : Intel GMA950 Integrated Graphics Controller

Audio Chipset : Realtek ALC880 High Definition Audio Controller

Lan Adapter: On-board Marvell 88e8052 Gigabit Ethernet Adapter

第二種

Product/Model: Acer Veriton M461

CPU: Intel Core 2 Quad CPU Q6600 2.4GHz

RAM: 2 GB

BIOS: AMI BIOS Veriton M461 Rev. R01-B0L

HBA : Intel Corporation 82801 (ICH9R) SATA RAID Controller

Intel Corporation PT IDER Controller

DVD : DVD D DH16D2S JA12 DVD-ROM

Hard Disk: WDC WD2500AAJS-22B4A0 , Serial ATA 250G

Video Adapter: Intel Corporation Integrated GMA3100 Graphics Controller

Audio Chipset: Realtek ALC 888S HD Audio Controller

Lan Adapter : Integrated Intel Corporation 82571B Gigabit Ethernet adapter

兩種電腦所使用的作業系統都是 Windows XP SP2，所需要安裝的軟體有 Python2.6、mpich2-1.3.2p1，完整安裝說明已放在附錄。

安裝完成之後進行例題的運算，運算主機分別以兩台 Veriton 6800(單 CPU)為一組用乙太網路將兩台電腦連線進行平行運算，另外一組則是以 Veriton M461 (雙 CPU)直接將這兩個 CPU 進行平行運算。

4.2 方法

使用區域預測修正法對於例題一到例題七進行運算，首先記錄只用一台單一 CPU 的電腦不進行平行的運算過程與時間，再將成是兩個 CPU 平行之後進行運算包括雙 CPU 電腦與兩台單 CPU 電腦結果記錄下來，最後將記錄下來的結果做比較。

程式所需要的資訊設定：

$m=3$; $s=2$	步數與階層
$h=\text{pow}(10,-3)*0.5$	步寬
$t_begin=0$	t 的起始點
$t_end=20$	t 的中點
$\text{soly}=\text{YInitial}(y_0,h)$	用 y_0 的值算 y_1
$\text{tspan}=\text{FixStepSize}(t_begin,t_end,h)$	利用 FixStepSize 函數明確把所有的 t 標出
$\text{num_of_tspan}=\text{len}(\text{tspan})$	算出全部的 t 的總數

另外，再將區域預測修正法的四個運算的矩陣參數變成 A、B、C、D 四個矩陣
以方便之後使用。

$$\begin{bmatrix} y_{i+1}^p \\ y_{i+2}^p \end{bmatrix} = [A] \begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix} + [B] \begin{bmatrix} 6h \cdot f(x_{i-1}, y_{i-1}) \\ 6h \cdot f(x_i, y_i) \end{bmatrix}$$

$$\begin{bmatrix} y_{i+1}^c \\ y_{i+2}^c \end{bmatrix} = [C] \begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix} + [D] \begin{bmatrix} 6h \cdot f(x_{i+1}, y_{i+1}^p) \\ 6h \cdot f(x_{i+2}, y_{i+2}^p) \end{bmatrix}$$

$$A = \begin{bmatrix} 5 & -4 \\ 28 & -27 \end{bmatrix} ; B = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 2 & 3 \end{bmatrix} ; C = \begin{bmatrix} \frac{5}{23} & \frac{28}{23} \\ -\frac{4}{23} & \frac{27}{23} \end{bmatrix} ; D = \begin{bmatrix} \frac{11}{69} & \frac{-2}{69} \\ \frac{6}{23} & \frac{1}{23} \end{bmatrix}$$

完成以上工作之後，直接讓程式執行進行區域預測修正法的運算。

t=[tspan[i-1],tspan[i]]	給定 t
y=[soly[i-1],soly[i]]	給定 y
estimatedt=[tspan[i+1],tspan[i+2]]	給定要預測的 t
predictory=predictor(A,B,y,t,h)	開始利用 y_i 與 y_{i-1} 對 y_{i+1} 還有 y_{i+2} 進行預測
soly.append(predictory[0])	將預測的 y_{i+1} 加入到解的 答的序列裡
soly.append(predictory[1])	將預測的 y_{i+2} 加入到解的 答的序列裡
correctory=corrector(C,D,y,predictory,estimatedt,h)	開始對 y_{i+1} 還有 y_{i+2} 進行 修正
soly[i+1]=correctory[0]	將修正的 y_{i+1} 加入到解的 答的序列裡
soly[i+2]=correctory[1]	將修正的 y_{i+2} 加入到解答 的序列裡

將每一個例題執行運算並將其結果列印出來

例題一：

真解式子：

$$y = e^{-1t}$$


```
def ExactSolution(t=0):
```

```
    lamda=1
```

```
    exactsol=[exp(-lamda*t)]
```

```
    return exactsol
```

常微分式子：

$$y^{\text{new}} = -y^{\text{past}}$$

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        lamda=1
```

```
        f.append([-lamda*y[i][0]])
```

```
    return f
```

例題二：

真解式子： $y = e^{\sin(t)}$

```
def ExactSolution(t=0):
```

```
    exactsol=[exp(sin(t))]
```

```
    return exactsol
```

常微分式子：

$$y^{\text{new}} = y^{\text{past}} \cos(t)$$

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        f.append([y[i][0]*cos(t[i])])
```

```
    return f
```

例題三：



真解式子： $y = \left[\frac{1}{1+t^2} \quad -2 \times \frac{t}{(1+t^2)^2} \right]$

def ExactSolution(t=0):

 exactsol=[1/(1+pow(t,2)), -2*t/pow((1+pow(t,2)),2)]

 return exactsol

常微分式子：

$y^{\text{new}} = [y_2^{\text{past}} \quad -2y_1^{\text{past}} \times (1 - 4t^2y_1^{\text{past}})]$

def fofODEs(y=[],t=[]):

 f=[]

 for i in xrange(len(t)):

 f.append([y[i][1],

 -2*pow(y[i][0],2)*(1-4*pow(t[i],2)*y[i][0]))]

 return f

例題四：

真解式子： $y = \frac{20}{1+19e^{-\frac{t}{4}}}$



def ExactSolution(t=0):

 exactsol=[20/(1+19*exp(-t/4))]

 return exactsol

常微分式子：

$y^{\text{new}} = \frac{y^{\text{past}}}{4 \left(1 - \frac{y^{\text{past}}}{20} \right)}$

def fofODEs(y=[],t=[]):

 f=[]

 for i in xrange(len(t)):

 f.append([y[i][0]/4.*(1-y[i][0]/20)])

 return f

例題五：

$$\text{真解式子：} y = \frac{1}{\sqrt{1+t}}$$

```
def ExactSolution(t=0):
```

```
    exactsol=[1/sqrt(1+t)]
```

```
    return exactsol
```

常微分式子：

$$y^{\text{new}} = \frac{-(y^{\text{past}})^3}{2}$$

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        f.append([pow(-y[i][0],3)/2])
```

```
    return f
```

例題六：

$$\text{真解式子：} y = \begin{bmatrix} (2 + \cos(t))\cos(t) & (2 + \cos(t))\sin(t) & \sin(t) \end{bmatrix}$$

```
def ExactSolution(t=0):
```

```
    exactsol=[(2+cos(t))*cos(t),(2+cos(t))*sin(t),sin(t)]
```

```
    return exactsol
```

常微分式子：

$$y^{\text{new}} = \begin{bmatrix} -y_2^{\text{past}} - \frac{y_1^{\text{past}} \times y_3^{\text{past}}}{\sqrt{(y_1^{\text{past}})^2 + (y_2^{\text{past}})^2}} & y_1^{\text{past}} - \frac{y_2^{\text{past}} \times y_3^{\text{past}}}{\sqrt{(y_1^{\text{past}})^2 + (y_2^{\text{past}})^2}} & \frac{y_1^{\text{past}}}{\sqrt{(y_1^{\text{past}})^2 + (y_2^{\text{past}})^2}} \end{bmatrix}$$

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        r=sqrt(pow(y[i][0],2)+pow(y[i][1],2))
```

```
        f.append([-y[i][1]-y[i][0]*y[i][2]/r,
```

$$y[i][0]-y[i][1]*y[i][2]/r,y[i][0]/r))$$

return f

例題七：

真解式子： $y = [\cos(t) \quad -\sin(t) \quad \sin(t) \quad \cos(t)]$

def ExactSolution(t=0):

 exactsol=[cos(t),-sin(t),sin(t),cos(t)]

 return exactsol

常微分式子： $y^{\text{new}} = \begin{bmatrix} y_2^{\text{past}} & \frac{-y_1^{\text{past}}}{\sqrt{(y_1^{\text{past}})^2 + (y_3^{\text{past}})^2}} & y_4^{\text{past}} & \frac{-y_3^{\text{past}}}{\sqrt{(y_1^{\text{past}})^2 + (y_3^{\text{past}})^2}} \end{bmatrix}$

def fofODEs(y=[],t=[]):

 f=[]

 for i in xrange(len(t)):

 r=sqrt(pow(y[i][0],2)+pow(y[i][2],2))

 f.append([y[i][1],-y[i][0]/pow(r,3),

 y[i][3],-y[i][2]/pow(r,3)])

 return f

單 CPU 運作時，七個例題都依照著順序城是順序一步一步進行，首先

$[A] \begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix} + [B] \begin{bmatrix} 6h \cdot f(t_{i-1} \quad y_{i-1}) \\ 6h \cdot f(t_i \quad y_i) \end{bmatrix}$ 矩陣內的 $f(t \quad y)$ 利用各個例題所給的 fofODEs

的函數求得預測值 $\begin{bmatrix} y_{i+1}^p \\ y_{i+2}^p \end{bmatrix}$ ，再將 $[C] \begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix} + [D] \begin{bmatrix} 6h \cdot f(t_{i+1} \quad y_{i+1}^p) \\ 6h \cdot f(t_{i+2} \quad y_{i+2}^p) \end{bmatrix}$ 值算出，矩陣內

$f(t \quad y)$ 是利用之前計算出來的預測值 $\begin{bmatrix} y_{i+1}^p \\ y_{i+2}^p \end{bmatrix}$ 帶入計算，最後求出修正後的 $\begin{bmatrix} y_{i+1}^c \\ y_{i+2}^c \end{bmatrix}$ 做

為其解，把此解與各例題給的 ExactSolution 真解函數運算結果相減的絕對值做為誤差值，並將誤差值與解儲存起來完成一組 $S = 2$ 、 $M = 3$ 的函數計算，持續把其它組的函數計算完成留下最大誤差與解如圖。

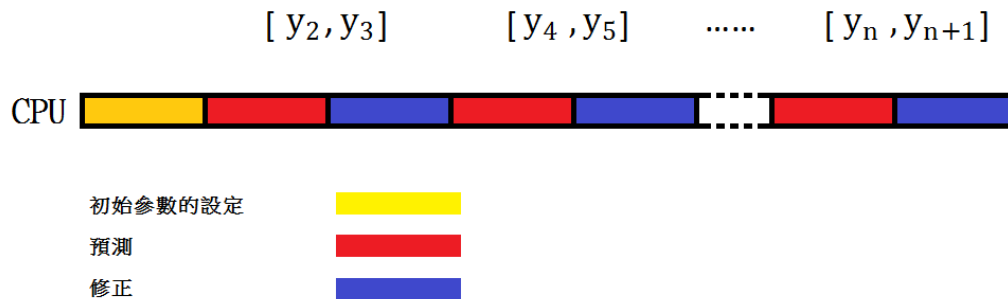


圖-5 PECE-在單 CPU 上

在平行之後得各 CPU，如同之前也是執行同樣七個例題，運作方式則不同於單 CPU 的運算，在計算之前要先把工作進行分配讓各 CPU 能在運算時取得自己的工作，CPU 取得工作分配的方式則是先利用 `Get_rank()` 這函數取的各 CPU 編號再利用 `if` 與 `elif` 來判斷 CPU 編號來給其工作，其程式碼如下：

```
mycomm = MPI.COMM_WORLD
```

```
myrank = mycomm.Get_rank()
```

```
if myrank == 0 :
```

(CPU0 的工作)

```
elif myrank == 1 :
```

(CPU1 的工作)



工作分配之後，許多變數、參數、物件經過單一 CPU 分開運算後的結果，只有單一個 CPU 擁有所以需要另外一個 CPU 進行傳送、接收，傳送接收利用 `send`、`recv` 兩函數進行，其程式碼如下：

CPU0

```
mycomm.send([y,MPI.FLOAT],1,1)
```

```
mycomm.send([t,MPI.FLOAT],1,2)
```

```
mycomm.recv([predictory,MPI.FLOAT],1,3)
```

CPU1

```
mycomm.recv([y,MPI.FLOAT],0,1)
```

```
mycomm.recv([t,MPI.FLOAT],0,2)
```

```
mycomm.send([predictory,MPI.FLOAT],0,3)
```

傳送接收的位置設定好之後便可運作並且平行，運作流程開始由 CPU0 傳送 y 與 t 給 CPU1，然後 CPU0 算矩陣 C 與 y 的相乘，同時 CPU1 則將 y 與 t 還有矩陣 $A \cdot B$ 帶入預測函數裡算出預測值並且傳送到 CPU0，得到預測值與 Cy 值得 CPU0 最後把修正值計算出來當作解，以上流程為一段函數得預測修正再將此動作放入迴圈裡進行完成全部的函數預測。如圖

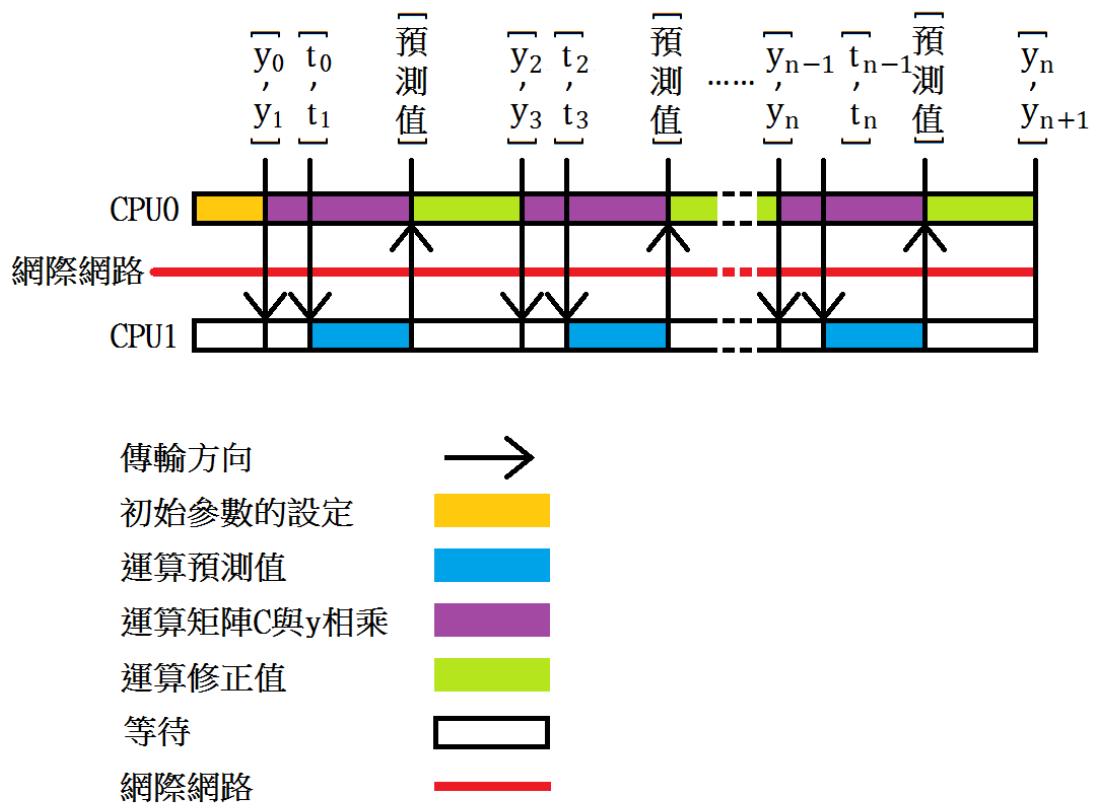


圖-6 PECE-在兩個 CPU 上

4.3 比較

由一個台電腦單一 CPU 未平行運作

題號	運算時間	最大誤差
例題一	0.711834854788	1.10555832154e-18
例題二	0.813653328789	3.40450334591e-10
例題三	1.03526565147	0.000303762981558, 7.82844619905e-05
例題四	0.741259940251	4.3517644599e-08
例題五	0.739818378763	7.76406716696e-13
例題六	1.19902854207	1.11890319232e-09, 2.68435584871e-09, 8.06434918843e-11
例題七	1.42250789867	3.12024204807e-08, 1.45567252696e-08, 1.16367976366e-08, 2.99500569345e-08

由雙 CPU 電腦進行平行運算

題號	運算時間	最大誤差
例題一	0.631384879381	1.10555832154e-18
例題二	0.805632562478	3.40450334591e-10
例題三	0.994352356787	0.000303762981558, 7.82844619905e-05
例題四	0.631946758251	4.3517644599e-08
例題五	0.736624905512	7.76406716696e-13
例題六	0.933512719463	1.11890319232e-09, 2.68435584871e-09, 8.06434918843e-11
例題七	1.207442512834	3.12024204807e-08, 1.45567252696e-08, 1.16367976366e-08, 2.99500569345e-08

由兩台單 CPU 電腦利用網路連結時的運算

題號	運算時間	最大誤差
例題一	14.25283582161	1.10555832154e-18
例題二	18.35724042568	3.40450334591e-10
例題三	25.69039345715	0.000303762981558, 7.82844619905e-05
例題四	15.23254675571	4.3517644599e-08
例題五	20.74574758792	7.76406716696e-13
例題六	30.77571247318	1.11890319232e-09, 2.68435584871e-09, 8.06434918843e-11
例題七	50.28793952464	3.12024204807e-08, 1.45567252696e-08, 1.16367976366e-08, 2.99500569345e-08



第五章 結論與展望

經過 Python 利用 mpi4py 在分散式電腦裡將 Predictor -Corrector method 的例題進行平行運算從第四章的比較中可以歸納出以下幾點：

1. 使用 MPICH2 與 Python 製作平行環境是非常簡單的，在計算上面是很有幫助的工具，而且 python 語法簡單並且是可以在 Windows 作業系統上執行。
2. 將程式以平行化的方式進行運算確實可以將消耗的時間縮短。
3. 在進行平行化的時候會因為網際網路傳輸時增加許多消耗的時間。
4. 實施平行化任務執行的編排時，必須儘量讓程式進行中每個 CPU 隨時都在運作避免處於閒置狀態，工作就可以更密集的完成，而工作耗時也可以變得更少。

上述結論，可以讓我們清楚的知道影響平行運算時間的因素不外乎是，程式部分哪些是可以加以平行化的，由其在數值運算上面經常會使用到矩陣相乘，對此方面未來可以在更進一步研究與加強，還有傳輸次數可以減少的儘量減少，網際網路傳輸在兩台 CPU 上會增加很多時間消耗，不只如此，平行程式撰寫的技巧更為重要，如此一來才能把平行化做到最佳狀態。

第六章 參考文獻

- [1]D. Voss ,S. ABBAS,"Block Predictor-Corrector Schemes for the Parallel Solution of ODES",Computers Math.Applic.Vol.33,No.6,,September 1996
- [2]陳志昇,"On Real Time Simulation of Block Predictor-Corrector Methods",國立中正大學應用數學研究所碩士論文,九十一年六月
- [3]Ming-Gong Lee, Rei-Wei Song, "A Family of Block Multistage-Multistep Method with Advanced Step-Points and Its Application to Numerical Solutions of Mildly stiff Ordinary Differential Equations",College of Engineering Chung Hua University
- [4] Siamak Mehrkanoon, Mohamed Suleiman, Zanariah Abdul Majid , Khairil Iskandar Othman, "Parallel Solution in Space of Large ODEs Using Block Multistep Method with Step Size Controller",European Journal of Scientific Research
- [5] Mathematics and Computer Science Division Argonne National Laboratory, "MPICH2 User's Guide",November 17, 2010
- [6] P.J. van der Houswen , B.P.Sommeijer, J.JB de Swart , "predictor -corrector method",Department of Numerical Mathematics NM-R9408 March 1944
- [7] Lisandro Dalcin , "MPI for Python Release 1.2.2",September 13, 2010
- [8] 蕭世文,"精通Python",2001年08月28日

第七章 附錄

附錄一(建立平行環境的安裝)

MPICH2 安裝完需要先使用提示命令字元模式將兩台平行的 Acer Veriton 6800 電腦金鑰設定成相同，並且把防火牆對於 MPICH2 的執行程式設定成例外不進行組擋，以避免在平行過程中資料傳輸時被防火牆攔截而產生錯誤，並且啟動 MPICH2 的 wmpiregister 進行使用者新增還有登入的動作如圖：

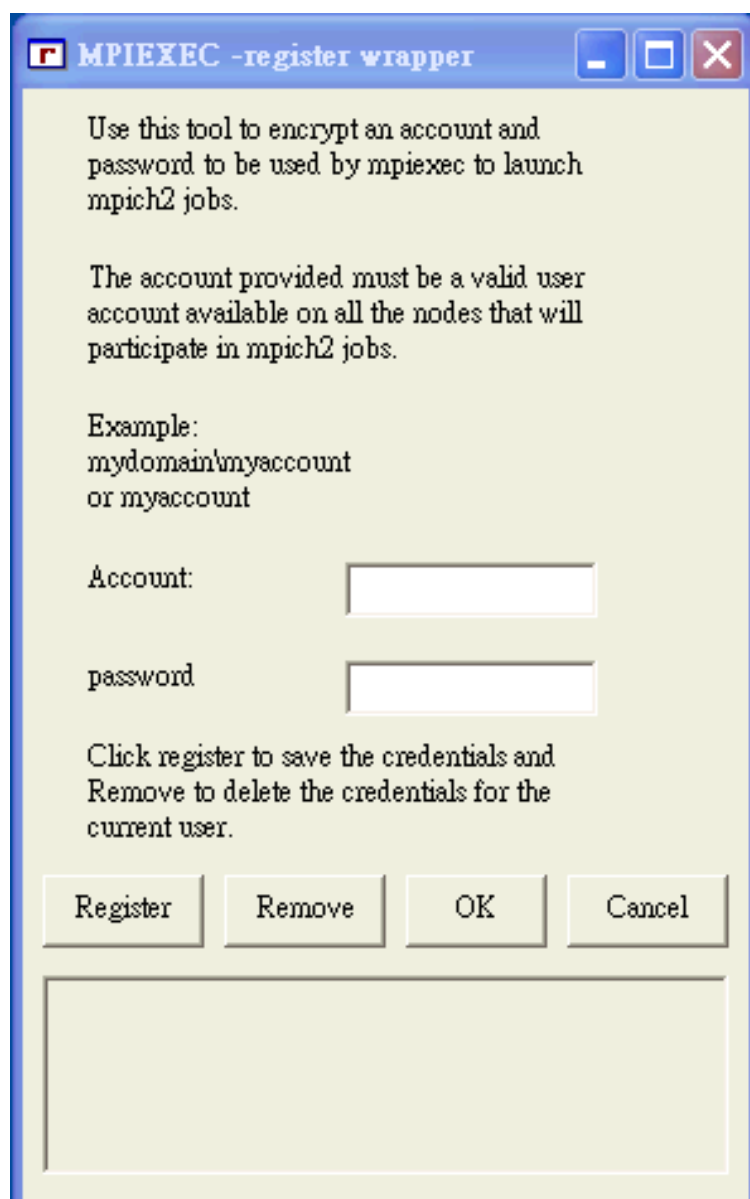


圖-7 wmpiregister

登入完成，再開啟 MPICH2 的 wmpiconfig 確認連線是否成功，成功則會顯是綠色失敗則是灰色。如圖：

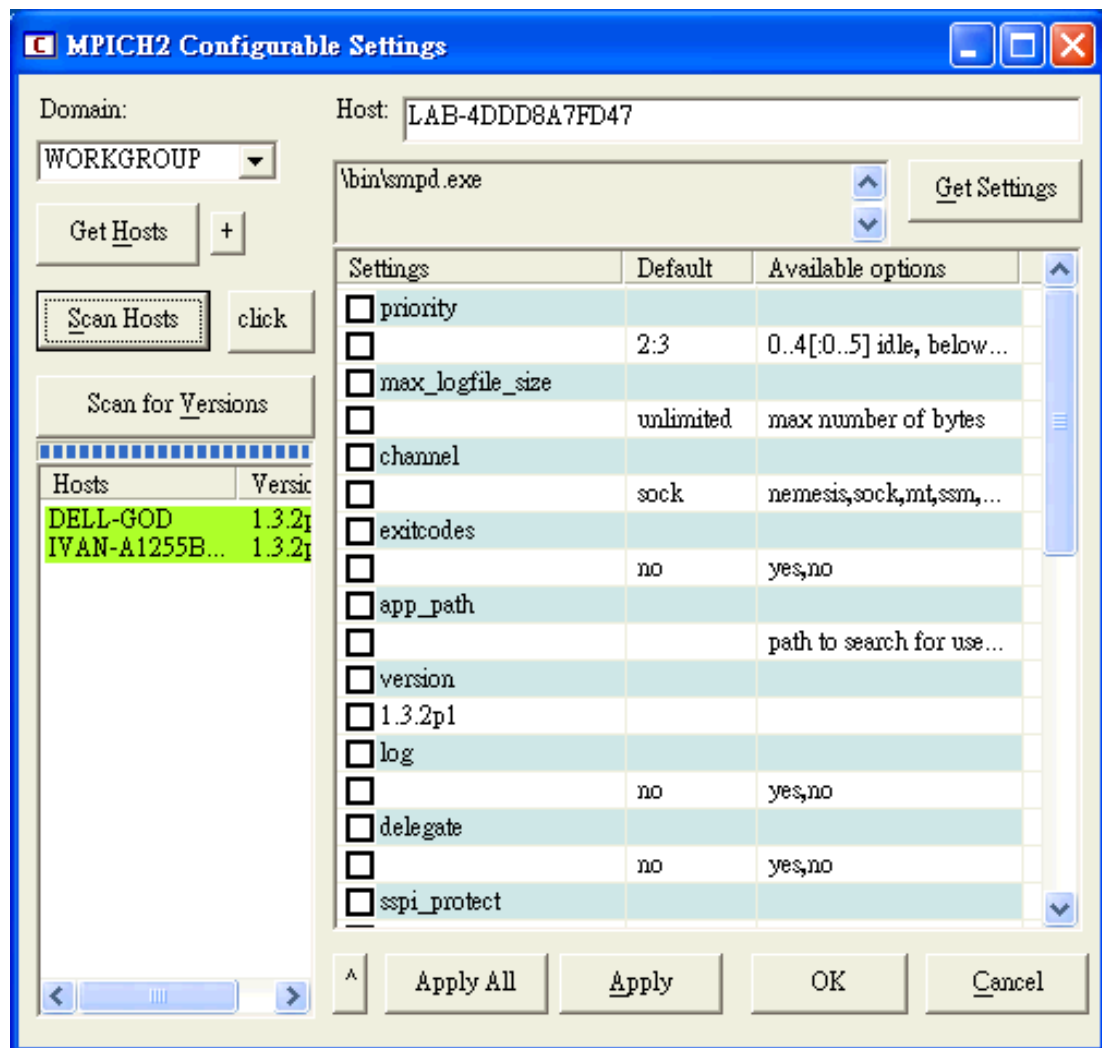
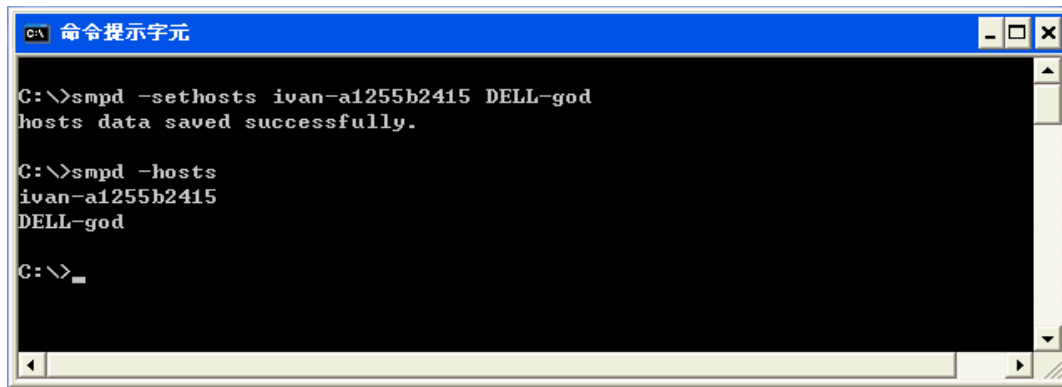


圖-8 wmpiconfig

確認後只代表連線沒有問題，不過運程式時還需要將兩台主機都加入到 host 的運作名單裡，設定方式由提示命令字元模式利用 `smpd -sethosts` 並將主機名稱加到後面看到 `hosts data saved successfully` 即完成。如圖



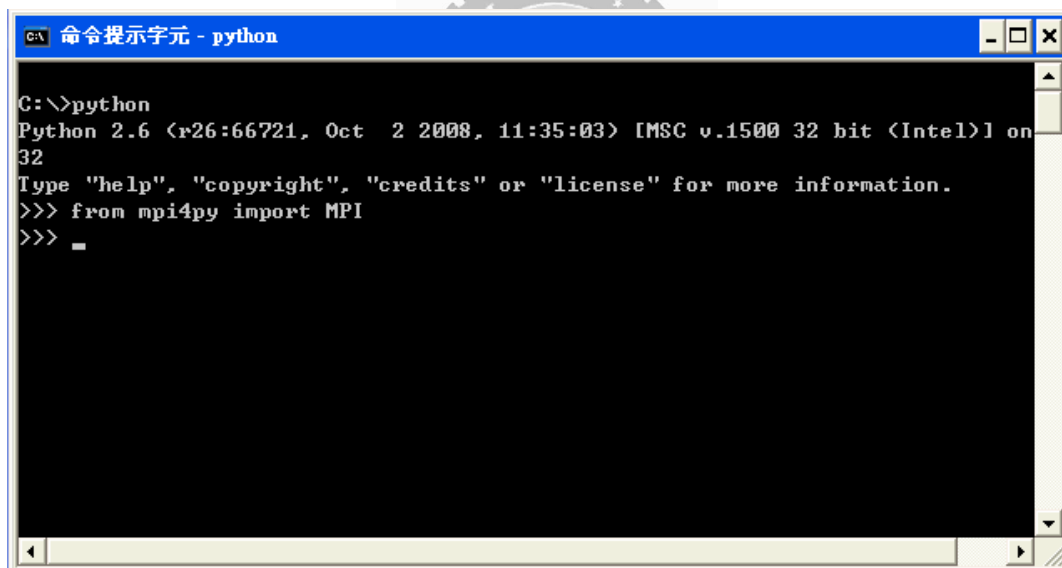
```
C:\>smpd -sethosts ivan-a1255b2415 DELL-god
hosts data saved successfully.

C:\>smpd -hosts
ivan-a1255b2415
DELL-god

C:\>_
```

圖-9 hosts data saved successfully

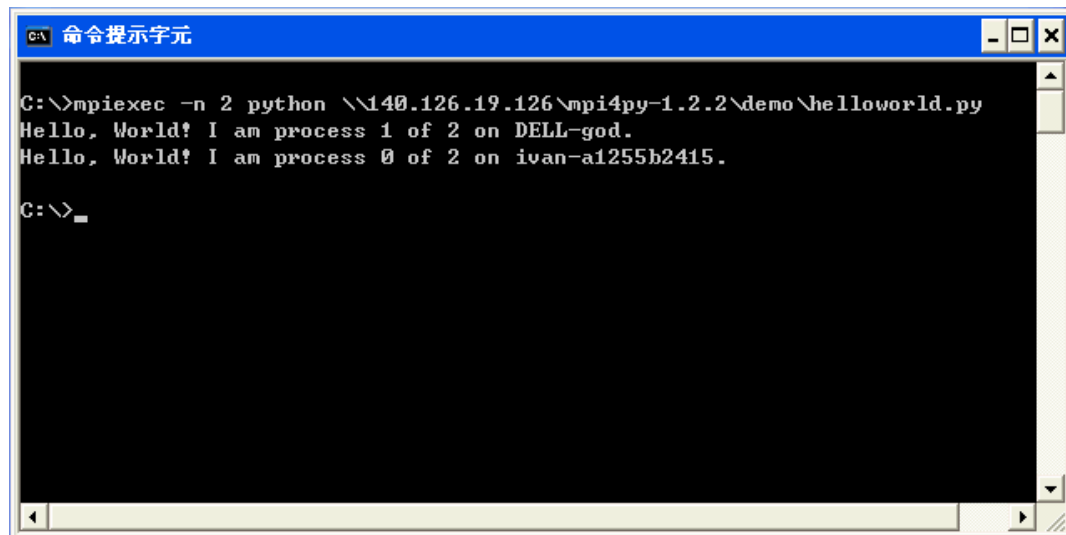
將 MPICH2 安裝與設定完成之後，要將 Python 可以進行 MPI 相關的程式，Python 則需要安裝 mpi4py 的套件，首先要先建立安裝的資料庫，使用提示命令字元模式執行 `python setup.py --build --mpi=mpich2-win` 讓 mpi4py 裡面產生一個 build 的資料夾再使用提示命令字元模式執行 `python setup.py --install` 進行安裝，利用 `import mpi4py` 裡的 MPI 函數來確定安裝是否成功。如圖



```
C:\>python
Python 2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)] on
32
Type "help", "copyright", "credits" or "license" for more information.
>>> from mpi4py import MPI
>>> _
```

圖-10 import mpi4py

為了確保整個系統能是完整沒有問題的，利用執行 `helloworld.py` 來確定兩台電腦都有進行 `helloworld` 的動作便可以進行各例題的計算。



```
C:\>mpiexec -n 2 python \\140.126.19.126\mpi4py-1.2.2\demo\helloworld.py
Hello, World! I am process 1 of 2 on DELL-god.
Hello, World! I am process 0 of 2 on ivan-a1255b2415.
C:\>_
```

圖-11 helloworld



附錄二(未平行的區域預測修正法的主程式)

```
#!/usr/bin/python

# -*- coding: utf-8 -*-

#

from math import *

def VectorAdd(vector1=[],vector2=[]):

    vectorsum=[]

    if vector2==[]:

        return vector1

    else:

        for i in xrange(len(vector1)):

            vectorsum.append(vector1[i]+vector2[i])

    return vectorsum

def MatrixAdd(matrix1=[],matrix2=[]):

    if matrix2==[]:

        return matrix1

    matrixlenght=len(matrix1)

    matrixsum=[]

    for i in xrange(matrixlenght):

        matrixsum.append(VectorAdd(matrix1[i],matrix2[i]))

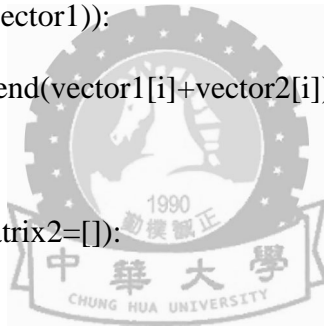
    return matrixsum

#

def Transpose(matrix=[]):

    row=len(matrix)

    column=len(matrix[0])
```



```

    return TransposeDifferent(matrix,row,column)

def TransposeDifferent(matrix=[],row=0,column=0):

    tranmatrix=[]

    for i in xrange(0,column):

        tranmvector=[]

        for j in xrange(0,row):

            tranmvector.append(matrix[j][i])

        tranmatrix.append(tranmvector)

    return tranmatrix

#

def MatrixCMultiply(c=0,matrix=[]):

    #print matrix

    matrix

    for i in xrange(len(matrix)):

        for j in xrange(len(matrix[0])):

            matrix[i][j]=c*matrix[i][j]

    return matrix

def VectorMultiply(vector1=[],vector2=[]):

    vectorproduct=[]

    for i in xrange(len(vector2)):

        product=[]

        for j in xrange(len(vector1)):

            product.append(vector1[j]*vector2[i])

        vectorproduct.append(product)

    return vectorproduct

def MatrixMultiply(matrix1=[],matrix2=[]):

```




```

matrix1=Transpose(matrix1)

vectorproduct=[]

matrixproduct=[]

for i in xrange(len(matrix2)):

    vectorproduct=VectorMultiply(matrix1[i],matrix2[i])

    matrixproduct=MatrixAdd(vectorproduct,matrixproduct)

matrixproduct=Transpose(matrixproduct)

return matrixproduct

#

def VectorDot(vector=[],y=[]):

    vectorproduct=[]

    product=0

    for i in xrange(len(vector)):

        product=product+vector[i]*y[i][0]

    vectorproduct.append(product)

    return vectorproduct

def MatrixVectorDot(matrix=[],y=[]):

    matrixproduct=[]

    for i in xrange(len(matrix)):

        matrixproduct.append(VectorDot(matrix[i],y))

    return matrixproduct

#-----step size

def FixStepSize(t_begin,t_end,h):

    i=t_begin

```



```

tspan=[]

while i<=t_end:

    tspan.append(i)

    i=i+h

return tspan

#-----initial value

def YInitial(y0=[],h=0):

    n=len(y0)

    initialY=[]

    j=0

    for i in xrange(s):

        if i==0:

            initialY.append(y0)

        else:

            initialY.append(ExactSolution(j))

        j=j+h

    return initialY

#-----predictor

def predictor(A=[],B=[],y=[],t=[],h=0):

    #A*y+6*h*B*ODEs(y,t)

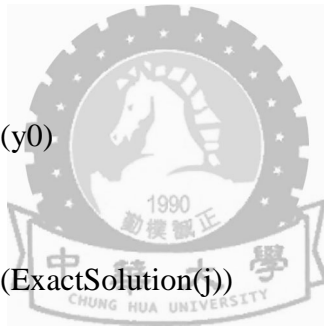
    predictor=MatrixAdd(MatrixMultiply(A,y),MatrixCMultiply(6*h,MatrixMultiply(B,
    fofODEs(y,t))))

    return predictor

#-----corrector

def corrector(C=[],D=[],y=[],estimatedy=[],estimatedt=[],h=0):

```



```

correctory=[]

#C*y+6*h*D*ODEs(predictor,t) or C*y+6*h*D*ODEs(corrector,t)

correctory=MatrixAdd(MatrixCMultiply(1,MatrixMultiply(C,y)),MatrixCMultiply(6*
h,MatrixMultiply(D,fofODEs(estimatedy,estimatedt))))

return correctory

```

#-----Ex.1

```
y0=[1]
```

```
def ExactSolution(t=0):
```

```
    lamda=1
```

```
    exactsol=[exp(-lamda*t)]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        lamda=1
```

```
        f.append([-lamda*y[i][0]])
```

```
    return f
```



#-----Ex.2

```
y0=[1]
```

```
def ExactSolution(t=0):
```

```
    exactsol=[exp(sin(t))]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):

    f=[]

    for i in xrange(len(t)):

        f.append([y[i][0]*cos(t[i])])

    return f
```

#-----Ex.3

y0=[1,0]

```
def ExactSolution(t=0):
```

```
    exactsol=[1/(1+pow(t,2)), -2*t/pow((1+pow(t,2)),2)]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        f.append([y[i][1], -2*pow(y[i][0],2)*(1-4*pow(t[i],2)*y[i][0])])
```

```
    return f
```



#Ex.4

y0=[1]

```
def ExactSolution(t=0):
```

```
    exactsol=[20/(1+19*exp(-t/4))]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        f.append([y[i][0]/4.*(1-y[i][0]/20)])
```

```
return f
```

```
#-----Ex.5
```

```
y0=[1]
```

```
def ExactSolution(t=0):
```

```
    exactsol=[1/sqrt(1+t)]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        f.append([pow(-y[i][0],3)/2])
```

```
    return f
```

```
#-----Ex.6
```

```
y0=[3,0,0]
```

```
def ExactSolution(t=0):
```

```
    exactsol=[(2+cos(t))*cos(t),(2+cos(t))*sin(t),sin(t)]
```

```
    return exactsol
```

```
def fofODEs(y=[],t=[]):
```

```
    f=[]
```

```
    for i in xrange(len(t)):
```

```
        r=sqrt(pow(y[i][0],2)+pow(y[i][1],2))
```

```
        f.append([-y[i][1]-y[i][0]*y[i][2]/r,y[i][0]-y[i][1]*y[i][2]/r,y[i][0]/r])
```

```
    return f
```

```
#-----Ex.7
```



```

y0=[1,0,0,1]

def ExactSolution(t=0):

    exactsol=[cos(t),-sin(t),sin(t),cos(t)]

    return exactsol

def fofODEs(y=[],t=[]):

    f=[]

    for i in xrange(len(t)):

        r=sqrt(pow(y[i][0],2)+pow(y[i][2],2))

        f.append([y[i][1],-y[i][0]/pow(r,3),y[i][3],-y[i][2]/pow(r,3)])

    return f

#-----PECE Main

m=3

s=2

h=pow(10,-3)

t_begin=0

t_end=0.05

soly=YInitial(y0,h)

tspan=FixStepSize(t_begin,t_end,h)

num_of_tspan=len(tspan)

A=[[5.,-4.],[28.,-27.]]

B=[[1/3.,2/3.],[2.,3.]]

C=[[-5/23.,28/23.],[-4/23.,27/23.]]

D=[[11/69.,-2/69.],[6/23.,1/23.]]

for i in xrange(1,num_of_tspan-2,s):

    t=[tspan[i-1],tspan[i]]

```



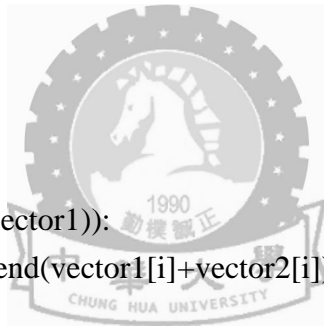
```
y=[soly[i-1],soly[i]]  
estimatedt=[tspan[i+1],tspan[i+2]]  
predictory=predictor(A,B,y,t,h)  
soly.append(predictory[0])  
soly.append(predictory[1])  
correctory=corrector(C,D,y,predictory,estimatedt,h)  
soly[i+1]=correctory[0]  
soly[i+2]=correctory[1]  
print soly
```



附錄三(平行的區域預測修正法的主程式)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
#mpiexec -n 2 python
\\140.126.19.158\mpi4py-1.2.2\demo\t3.py>c:\mpi4py-1.2.2\demo\t3.txt
import sys
from mpi4py import MPI
import numpy as N
from math import *
import time

def VectorAdd(vector1=[],vector2=[]):
    vectorsum=[]
    if vector2==[]:
        return vector1
    else:
        for i in xrange(len(vector1)):
            vectorsum.append(vector1[i]+vector2[i])
        return vectorsum
def MatrixAdd(matrix1=[],matrix2=[]):
    if matrix2==[]:
        return matrix1
    matrixlenght=len(matrix1)
    matrixsum=[]
    for i in xrange(matrixlenght):
        matrixsum.append(VectorAdd(matrix1[i],matrix2[i]))
    return matrixsum
#
def Transpose(matrix=[]):
    row=len(matrix)
    column=len(matrix[0])
    return TransposeDifferent(matrix,row,column)
def TransposeDifferent(matrix=[],row=0,column=0):
    tranmatrix=[]
```




```

    for i in xrange(0,column):
        tranmvector=[]
        for j in xrange(0,row):
            tranmvector.append(matrix[j][i])
        tranmatrix.append(tranmvector)
    return tranmatrix

#
def MatrixCMultiply(c=0,matrix=[]):
    #print matrix
    matrix
    for i in xrange(len(matrix)):
        for j in xrange(len(matrix[0])):
            matrix[i][j]=c*matrix[i][j]
    return matrix

def VectorMultiply(vector1=[],vector2=[]):
    vectorproduct=[]
    for i in xrange(len(vector2)):
        product=[]
        for j in xrange(len(vector1)):
            product.append(vector1[j]*vector2[i])
        vectorproduct.append(product)
    return vectorproduct

def MatrixMultiply(matrix1=[],matrix2=[]):
    matrix1=Transpose(matrix1)
    vectorproduct=[]
    matrixproduct=[]
    for i in xrange(len(matrix2)):
        vectorproduct=VectorMultiply(matrix1[i],matrix2[i])
        matrixproduct=MatrixAdd(vectorproduct,matrixproduct)
    matrixproduct=Transpose(matrixproduct)
    return matrixproduct

#
def VectorDot(vector=[],y=[]):
    vectorproduct=[]
    product=0
    for i in xrange(len(vector)):
        product=product+vector[i]*y[i][0]
    vectorproduct.append(product)

```

```

        return vectorproduct
def MatrixVectorDot(matrix=[],y=[]):
    matrixproduct=[]
    for i in xrange(len(matrix)):
        matrixproduct.append(VectorDot(matrix[i],y))
    return matrixproduct

#-----step size
def FixStepSize(t_begin,t_end,h):
    #    i=t_begin
    #    tspan=[]
    #    while i<=t_end:
    #        tspan.append(i)
    #        i=i+h
    #    return tspan
#-----initial value
def YInitial(y0=[],h=0):
    n=len(y0)
    initialY=[]
    j=0
    for i in xrange(s):
        if i==0:
            initialY.append(y0)
        else:
            initialY.append(ExactSolution(j))
        j=j+h
    return initialY
#-----predictor
def predictor(A=[],B=[],y=[],t=[],h=0):
    #A*y+6*h*B*ODEs(y,t)

    predictory=MatrixAdd(MatrixMultiply(A,y),MatrixCMultiply(6*h,MatrixMultiply(B,
    fofODEs(y,t))))
    return predictory
#-----corrector
def corrector(D=[],Cy=[],estimatedy=[],estimatedt=[],h=0):
    correctory=[]

```



```

#C*y+6*h*D*ODEs(predictor,t) or C*y+6*h*D*ODEs(corrector,t)

correctory=MatrixAdd(MatrixCMultiply(1,Cy),MatrixCMultiply(6*h,MatrixMultiply
(D,fofODEs(estimatedy,estimatedt))))
    return correctory

#-----
#Ex.1
y0=[1]
def ExactSolution(t=0):
    lamda=1
    exactsol=[exp(-lamda*t)]
    return exactsol
def fofODEs(y=[],t=[]):
    f=[]
    for i in xrange(len(t)):
        lamda=1
        f.append([-lamda*y[i][0]])
    return f

#-----PECE Main
m=3
s=2
h=pow(10,-3)
t_begin=0.
t_end=20.
soly=YInitial(y0,h)
#tspan=FixStepSize(t_begin,t_end,h)
#num_of_tspan=len(tspan)
num_of_tspan=int((t_end-t_begin)/h+1)
mycomm = MPI.COMM_WORLD
name = MPI.Get_processor_name()
myrank = mycomm.Get_rank()
y=[soly[0],soly[1]]
t=[0.,0.+h]
if myrank == 0:

```



```

C=[[-5/23.,28/23.],[-4/23.,27/23.]]
D=[[11/69.,-2/69.],[6/23.,1/23.]]
predictory=[[0],[0]]
elif myrank == 0:
    A=[[5.,-4.],[28.,-27.]]
    B=[[1/3.,2/3.],[2.,3.]]
#wt = MPI.Wtime()
timepiece=time.clock()
for i in xrange(1,num_of_tspan-2,s):
    #sys.stdout.write("i= %d ! I am process %d on %s.\n" % (i, myrank, name))
    #print i
    if myrank == 0 :
        t=[(i+1)*h,(i+2)*h]
        y=[soly[i-1],soly[i]]
        mycomm.send([y,MPI.FLOAT],1,1)
        mycomm.send([t,MPI.FLOAT],1,2)
        mycomm.recv([predictory,MPI.FLOAT],1)
        Cy=MatrixMultiply(C,y)
        estimatedt=[(i+3)*h,(i+4)*h]
        correctory=corrector(D,Cy,predictory,estimatedt,h)
        soly.append(correctory[0])
        soly.append(correctory[1])
    elif myrank ==1 :
        mycomm.recv([y,MPI.FLOAT],0,1)
        mycomm.recv([t,MPI.FLOAT],0,2)
        predictory=y
        mycomm.send([predictory,MPI.FLOAT],0)
if myrank ==0:
timepiece=time.clock()-timepiece
wt = MPI.Wtime() - wt
print soly
print wt

```

附錄四(程式中運算的資料類型與 C 語言的對照)

MPI data type	C data type	description
MPI_CHAR	signed char	1-byte character
MPI_SHORT	signed short int	2-byte integer
MPI_INT	signed int	4-byte integer
MPI_LONG	signed long int	4-byte integer
MPI_UNSIGNED_CHAR	unsigned char	1-byte unsigned character
MPI_UNSIGNED_SHORT	unsigned short int	2-byte unsigned integer
MPI_UNSIGNED	unsigned int	4-byte unsigned integer
MPI_UNSIGNED_LONG	unsigned long int	4-byte unsigned integer
MPI_FLOAT	float	4-byte floating point
MPI_DOUBLE	double	8-byte floating point
MPI_LONG_DOUBLE	long double	8-byte floating point
MPI_PACKED		