國家高速網路與計算中心

R-MPI 技術報告

專案助理研究員 劉嘉言

2008/10/7

內容

一、 前言

本文件主要是給那些想要使用 R、R-MPI 來計算或是統計大量資料的使用者參考，透過這份文件描述的內容，R 的程式設計者可以將所設計出來的程式藉由 R-MPI 在多台電腦上來執行，達到節省執行時間的目的；而這其中所需付出的代價可能只是僅僅多出不到 10% 的程式碼，但是卻可以帶來節省 60% 以上的程式執行時間的效益。

一開始我們會先簡單地介紹 MPI、R、R-MPI 等名詞，接著說明應該如何建立一個 R/R-MPI 的計算平台，之後再進一步地描述其運作方式及原理，並且探討 R-MPI 常用的一些函式庫，同時，再以範例來解釋如何撰寫一個 R/R-MPI 的程式，最後，驗證 R/R-MPI 平行處理後所帶來的效益。

1. 名詞簡介

I. MPI

MPI (Message Passing Interface) [1][2] 顧名思義就是一個傳遞訊息的界面。目前已經發展到 MPI-2.0，同時亦具有許多不同的實作(with GPL license)可供自由使用。在我們所欲建立的 R/R-MPI 平台中，必須利用 MPI 所定義的界面在不同的電腦之間互相傳遞交換訊息，進而達到在多台電腦上平行運算的目的。而我們所採用的 MPI implementation 爲 mpich2 [3]。

II. R

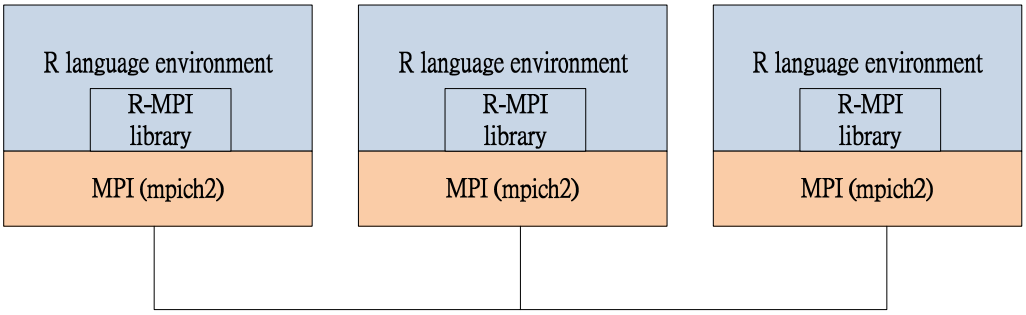
R [4] 是一種專門用於統計的語言，因它對於大量的資料處理，像是迴圈的運算，具有非常好的效能，使得 R 語言在科學運算中的應用也越來越爲廣泛。在 R 語言的套件中，有內建了許多統計相關的函式，使得我們可以在很短的時間內 (< 1 秒)便可以計算出數千個甚至是數萬的數據中的標準差、中位數等等。我們會在後面的章節簡單地介紹 R 的資料結構以及其語法。

III. R-MPI

R-MPI [5] 是一個架構於 R 與 MPI (eg, mpich, LAM, mpich2) 之上的自由軟體，它是由 Department of mathematics and statistic at Acadia University 所開發。 R-MPI 提供一個介面並且利用底層預先設定安裝好的的 MPI，使得不同電腦之間可以互相交換、傳遞 R 的物件或是其他相關的訊息、資料等等，進而達到將 R 程式平行運算的目的。

2. 系統架構與運作原理

R-MPI 可以視爲是 R 裡面的一個 library，而這個 library 還必須仰賴下層的 MPI 才能夠正常地運作。如圖表 1 所示，R-MPI library 安插在 R 的環境中，並且提供許多 R-MPI 的相關函式供使用者在設計 R 程式的時候可以將需要運算的需求透過底下的 MPI 將 R 的物件或是其他相關的訊息傳送到其他的電腦；其它的電腦在收到的時候，一樣透過 MPI 將訊息收進來，再透過 R-MPI 把 R 的物件或是訊息送到 R 的環境中進行平行處理。



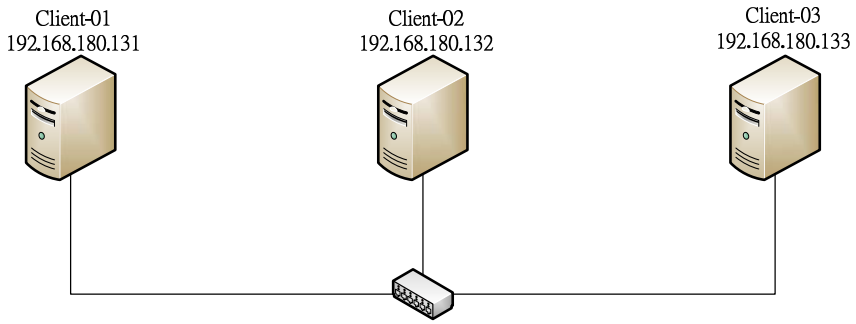
圖表 2

I. 環境介紹

以下我們將使用三台機器作爲安裝 R/R-MPI 的範例，圖表 3 是這三台機器的規格、IP 設定。以及其連接的方式。圖表 4 則是這三台機器連接的示意圖。

hostname	IP address	OS	HD	RAM
Client-01	192.168.180.131	Ubuntu 7.04	5GB	128MB
Client-02	192.168.180.132	Ubuntu 7.04	5GB	128MB
Client-03	192.168.180.133	Ubuntu 7.04	5GB	128MB

圖表 5



圖表 6

II. MPI 的安裝步驟

MPI 目前提供數個 GPL 的 implementations 可供使用，像是 mpich、LAM、mpich2、OpenMPI 等等都已經相當成熟。除了 OpenMPI 目前已知道對於 R-MPI 的支援尚有問題外[6]，其餘的 implementation 對於 R-MPI 的支援都可以正常運作。以下我們將以 Linux-based 的作業系統 (Ubuntu 7.04)平台爲例，說明如何安裝 mpich2。

i. 設定主機名稱

以 client-01 爲例，首先請先修改 /etc/hosts 中 IP 位址與主機名稱的對應關係，特別要注意的是所有欲參與平行運算的機器都必須擁有相同的 /etc/hosts 設定，這是因爲 mpich2 在啓動時會根據主機名稱尋找所對應的 IP。

```
root@client-01:~# cat > /etc/hosts << "EOF"
> 127.0.0.1 localhost
> 192.168.180.131 client-01
> 192.168.180.132 client-02
> 192.168.180.133 client-03
> EOF
```

同時請注意務必將 127.0.0.1 hostname 這行刪除，否則 mpi 的啓動將會出現中斷，這是因爲 mpi 在尋找 client-01 時會找到 loopback address 也就是 127.0.0.1，但這並非一個有效的 IP 位址，故會造成 MPI

啓動時的錯誤，所以在各個機器的 /etc/hosts 中，請務必刪除 127.0.0.1 hostname。

ii. 下載並安裝 mpich2

在本文件中，我們將以 mpich2-1.0.7rc1 這個版本爲範例，首先必須下載 mpich2-1.0.7rc1.tar.gz ，接著再進行解壓縮的動作 (解壓縮到任何一目錄都可以)。

```
root@client-01:~# wget
http://www.mcs.anl.gov/research/projects/mpich2/downloads/tarballs/m
pich2-1.0.7rc1.tar.gz
root@client-01:~# tar -zxvf mpich2-1.0.7rc1.tar.gz
cd mpich2-1.0.7rc1/
```

進入 mpich2-1.0.7rc1 目錄之後，接著就可以開始設定 mpich2 所需安裝的位置以及參數等。在這個範例中，我們假設安裝的路徑是 /opt/mpich2 ，參數爲 prefix=/opt/mpich2 ，其餘爲編譯時的參數。

```
root@client-01:~/mpich2-1.0.7rc1# ./configure CFLAGS="-fPIC"
CXXFLAGS="-fPIC" FFLAGS="-fPIC" prefix=/opt/mpich2
root@client-01:~/mpich2-1.0.7rc1# make
root@client-01:~/mpich2-1.0.7rc1# make install
```

若是在上述的編譯過程中有問題，也許是您尚未安裝編譯的工具，在 Ubuntu 7.04 上面可使用 apt-get 來安裝編譯時所需要的工具。

```
root@client-01:~/mpich2-1.0.7rc1# apt-get install build-essential
```

到此算是已經安裝完成了，可以使用 which 指令來檢查一下 mpich2 的執行檔的位置。

```
root@client-01:~/mpich2-1.0.7rc1# which mpd
/opt/mpich2/bin/mpd
root@client-01:~/mpich2-1.0.7rc1# which mpicc
/opt/mpich2/bin/mpicc
root@client-01:~/mpich2-1.0.7rc1# which mpiexec
/opt/mpich2/bin/mpiexec
root@client-01:~/mpich2-1.0.7rc1# which mpirun
/opt/mpich2/bin/mpirun
```

若是無法找到相對應的執行檔位置，那麼很可能是您的環境變數尚未設定好，可以使用 export 命令將環境變數更新；或是將環境變數加入 ~/.bashrc 檔中。

```
root@client-01:~/mpich2-1.0.7rc1# export PATH="$PATH:/opt/mpich2/bin"
```

確認安裝無誤之後，請新增一個 mpd.hosts 的檔案，這個檔案的檔名可以自由選擇，但是必須記得這個檔案的位置，在每一台機器上面都必須設定相同位置、內容的 mpd.hosts 檔。mpd.hosts 的目的在於提供 mpich2 知道有哪些節點要加入 mpi 的運算環境中。

```
root@client-01:~/mpich2-1.0.7rc1# touch mpd.hosts
root@client-01:~/mpich2-1.0.7rc1# cat > mpd.hosts << "EOF"
> client-01
> client-02
> client-03
> EOF
```

最後，如果您是用 root 執行 mpich2 的話，那麼就必須在每台機器的 /etc 目錄下新增一個 mpd.conf 檔；反之，若您打算使用一般使用者執行 mpich2 的話，就必須在該使用者的家目錄 (/home/username/) 底下新增一個 .mpd.conf 檔 (沒錯，這是一個隱藏檔)。

這個檔案的內容只有一行密碼，此密碼是用來讓每一台機器可以認可彼此。擁有相同密碼的機器就會形成一個 MPI 的環境。最後再更改這個密碼檔的權限即可。

```
root@client-01:~/mpich2-1.0.7rc1# touch /etc/mpd.conf
root@client-01:~/mpich2-1.0.7rc1# cat > /etc/mpd.conf << "EOF"
> secretword=this_is_password
> EOF
root@client-01:~/mpich2-1.0.7rc1# chmod 600 /etc/mpd.conf
```

iii. 設定 ssh 連線的金鑰

mpich2 將會透過 ssh 連線來連結每一台機器，因此，我們必須將 ssh 在建立連線時，將要求輸入密碼的提示給取消掉，方法是在每一台機器上建立一把彼此都認可的金鑰，有了這把金鑰的存在，每台機器彼此之間就可以透過 ssh 連線而不需要輸入密碼。

在本文件中，我們將以使用者 root 爲例子來建立 ssh 連線過程中所需要的金鑰，過程中所出現的提示可以不需特別設定，直接按下 Enter 鍵即可。

```
root@client-01:~/mpich2-1.0.7rc1# ssh-keygen -t dsa
```

```
Generating public/private dsa key pair.
Enter file in which to save the key (/root/.ssh/id_dsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_dsa.
Your public key has been saved in /root/.ssh/id_dsa.pub.
The key fingerprint is:
d7:ad:ce:0f:e2:d8:16:3d:92:10:b8:b3:7f:7b:68:d9 root@client-01
root@client-01:~/mpich2-1.0.7rc1#
```

完成上述步驟之後，在 client-01 裡面的 /root/.ssh 裡面就會出現一個檔案爲 id_dsa.pub，這是金鑰的目錄名稱，接著我們將金鑰目錄更改命名爲 authorized_keys ，並且將這個金鑰目錄複製到每一台機器上，同樣位於 /root/.ssh 這個目錄中 (若是一般使用者就是 /home/username/.ssh 目錄)。

```
root@client-01:~/mpich2-1.0.7rc1# cd /root/.ssh/
root@client-01:~/.ssh# cp ./id_dsa.pub ./authorized_keys
root@client-01:~/.ssh# cd ..
root@client-01:~# scp /root/.ssh/* root@client-02:/root/.ssh/
root@client-02's password:
authorized_keys                               100%  610     0.6KB/s
00:00
id_dsa                                        100%  668     0.7KB/s
00:00
id_dsa.pub                                    100%  610     0.6KB/s
00:00
known_hosts                                   100% 4420     4.3KB/s
00:00
root@client-01:~# scp /root/.ssh root@client-03:/root/.ssh/
root@client-03's password:
authorized_keys                               100%  610     0.6KB/s
00:00
id_dsa                                        100%  668     0.7KB/s
00:00
id_dsa.pub                                    100%  610     0.6KB/s
00:00
known_hosts                                   100% 4420     4.3KB/s
00:00
```

完成上述步驟之後，可以使用 ssh 指令連線測試，驗證每一台機器之間是否不需輸入密碼即可建立 ssh 連線。

iv. 啓動/停止與測試

最後我們可已下達一些指令來測試看看 mpich2 是否可以正常啓

動。一般來說，啓動的指令爲 mpdboot，參數 “-n” 後面接的是 slaves 的數目，在本範例中爲 3 台 slaves 加上 1 台 server，因此 –n 後面要接的數目應該是 3。參數 “-f” 後面接的是 mpd.hosts 檔案的位置，在前面我們曾經提過 mpd.hosts 裡面記錄的是欲參與 mpich2 運算的節點名稱。

```
root@client-01:~# mpdboot -n 3 -f /root/mpich2-1.0.7rc1/mpd.hosts
```

執行後，如果沒有任何錯誤訊息出現，即表示 mpich2 已經正常地啓動了，接下來可以輸入指令觀察目前參與 MPI 運算的節點是否和 mpd.hosts 裡面記錄的一樣。

```
root@client-01:~# mpdtrace
client-01
client-03
client-02
```

最後我們可以執行簡單的測試，看看每個節點之間傳遞訊息所需花費的時間。

```
root@client-01:~# mpdringtest
time for 1 loops = 0.00142002105713 seconds
root@client-01:~# mpdringtest 3
time for 3 loops = 0.00336718559265 seconds
root@client-01:~# mpdringtest 100
time for 100 loops = 0.108873844147 seconds
```

欲停止 mpich2 可以輸入 mpdallexit 的指令。

```
root@client-01:~# mpdallexit
```

III. R 的安裝步驟

由於 Ubuntu 7.04 的 apt 儲藏庫中已經內建了 R 的相關套件，因此只要利用 apt-get 即可完成 R 的安裝。值得注意的是 apt 儲藏庫中的 R 的相關套件，預設是必須使用 gcc-4.2 以上的版本編譯，如果不想重新安裝 gcc-4.2 可以直接把 gcc link 到 gcc-4.2 即可。如果想要使用 source 安裝，可以參閱官方網站的說明 [7] 。

```
root@client-01:~# apt-get install r-base-core
```

IV. R-MPI 的安裝步驟

透過 R 的指令可快速地安裝 R-MPI，因此在安裝 R-MPI 之前，建議先將 R 安裝好。R-MPI 的安裝非常簡單只要先下載再使用 R 的指令安裝即可。

```
root@client-01:/opt# wget
http://www.stats.uwo.ca/faculty/yu/Rmpi/download/linux/Rmpi_0.5-5.tar.gz
root@client-01:/opt# R CMD INSTALL Rmpi_0.5-5.tar.gz
--configure-args=--with-mpi=/opt/mpich2
```

若是使用 R 指令安裝時，請務必加上 –configure-args 的參數，這個參數的目的是爲了讓 R-MPI 知道 mpi 所安裝的所在位置，當 R-MPI 啓動的時候就會去找 mpi 對應的執行檔幫忙傳送訊息。

V. 測試

上述步驟接順利完成後，就可以開始進行一些測試，看看 R/R-MPI 的計算平台是否可以正常的運作。首先，啓動 mpich2，接著執行 R-MPI 官方網站所提供的範例。

```
root@client-01:/opt# mpdboot -n 3 -f /root/mpich2-1.0.7rc1/mpd.hosts
root@client-01:/opt# mpdtrace
client-01
client-03
client-02
```

確認 mpi 已經 ready。下載 R-MPI 官方網站的範例程式並執行。

```
root@client-01:/opt# wget
http://ace.acadiau.ca/math/ACMMaC/Rmpi/task_pull.R
root@client-01:/opt# mpirun -np 1 R --slave CMD BATCH task_pull.R
```

由於 R-MPI 是建構在 mpich2 上，因此我們執行 mpirun 時，後面再接 R 的指令 (當成 mpich2 的參數) 便可以成功地透過 R-MPI 將所需要平行運算的任務分配出去。參數 ”-np” 表示 number of processors，這邊由於每台電腦只有 1 個 processor，因此後面接 1；接著再加入 R 的執行命令，並且讓它在背景執行。順利執行完畢之後，會產生一個 Rplots.ps 檔，這是範例程式最後生成的檔案，到這裡就表示 R/R-MPI 平台運作正常。

二、 R-MPI 相關函式功能

在撰寫 R-MPI 程式時，務必先瞭解其相關函式的定義與功能，透過這些 R-MPI 內建的函式，可以協助我們迅速地寫出自己想要平行化的 R 程式。全部的 R-MPI 函式庫可以在 R-MPI 的官網查詢 [8]，在這裡我們只列出一些比較常用、比較重要的函式。

1. library("Rmpi")

將 Rmpi 的 package 載入，讓我們可以使用 MPI 的 function。所以在使用任何 MPI functions 前必須先執行這個指令。

2. mpi.spawn.Rslaves(nslaves=#)

會產生出 # 個 processes，如果沒有任何參數，則會產出這個 MPI 環境中最大數量的 nodes。

3. mpi.close.Rslaves()

將所有的 slaves 關閉，建議在結束程式前必須執行此程式。

4. mpi.quit([saving=yes/no])

將 Rmpi 所分配的資源清除，並且呼叫 R 的 quit function，我們必須使用這個 function 離開程式。

5. mpi.comm.size()

回傳 processes 的個數，假設將每個 node 視爲一個 process 的話，那麼這個數目就是 slaves 個數加上 1 個 master。

6. mpi.comm.rank()

回傳每個 process 所代表的數字，假設將每個 node 視爲一個 process 的話，那麼就是每個 node 的 ID。

7. mpi.bcast.Robj2slave(object)

將一個 R object(number, string or list) 或是 function 送給所有的 slaves 。

8. mpi.send.Robj(object,destination,tag)

將一個 R object(number, string or list) 送給 destination ， destination 是就是 nodes 的 rank 。舉例來說， mpi.send.Robj(object,0,tag) ，會送到 master ， mpi.send.Robj(object,10,tag) 會送到 rank = 10 的 slave 。tag 則是用來識別訊息的標籤，有些訊息是指示 master 繼續傳送任務；有些訊息是指示 master 收集儲存 slaves 運算的結果等等。透過 tag 的識別，可以讓 R-MPI 的程式可以更彈性化地設計，更有效率地執行。

9. object <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())

使用這個 function 的 process 會從呼叫它的 process 的 queue 中取得訊息，如果 queue 裡沒有資料，則 process 會一直等到接到訊息爲止。

10. info <- mpi.get.sourcetag()

當在 mpi.recv.Robj 後使用這個 function ，會回傳兩個整數，一個是訊息所傳送的 node 編號，一個是它的 tag 。

11. mpi.bcast.cmd("R code")

讓所有的 slave processes 執行 R code ， 這個 function 不會等待 slave 執行完畢。

12. results <- mpi.remote.exec("R code")

讓所有的 slave processes 執行 R code ，並且回傳結果，這個 function 會等待所有的 slave 執行完畢後才回傳結果。

三、 三種撰寫 R-MPI 程式的方法

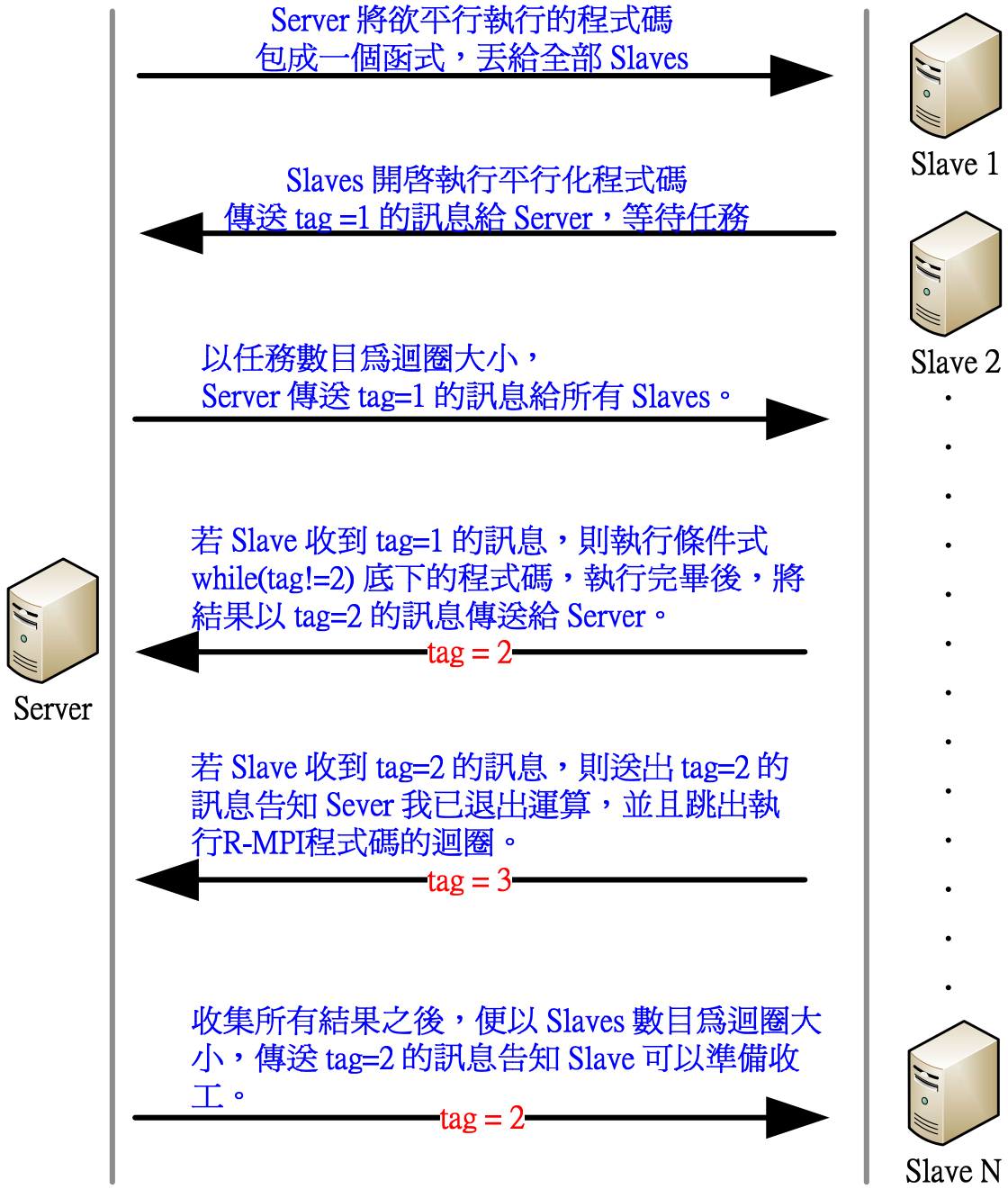
根據 R-MPI 的官網所描述，R-MPI 平行處理的程式撰寫上，可以分爲三種方式:

1. Brute Method

這是最簡單的方式，我們把 Task 當成是迴圈，假設今天有 10 個迴圈，要同時進行平行處理，那麼就必須要有 10 台 slaves，把所謂的 Task(任務)， 平均分配到 10 個不同的 slaves 上面執行。這種方式的優點是寫法最簡單，所需要外加的程式碼也會最少；但缺點是擴充性(scalability)也最差，一旦任務的數目大於 slaves 的數目，那麼此種方式就不適合使用。由於甚少有情況是任務數目剛好等於 slaves 的數目，因此這種寫法雖然簡單但是並不實用，在這裡就不多作介紹。

2. Task Push

顧名思義，Task push，就是將任務送(push)到 slaves 上去處理，直到所有任務都處理完畢爲止。這個方式改善了 Brute Method 中 --- "任務數目必須等於 slaves 數目"的這個缺點；也就是說 Task Push 可以處理任務數目大於 slaves 數目的情況。但是仍然不夠完美，最主要的原因是，在一般的情況下，不見得每一台 slaves 的運算能力都是一樣的，這個方法最大的缺點就是當 slaves 的運算能力不一時，運算能力較強的機器必須花費多餘的時間來等待運算能力較弱的機器。

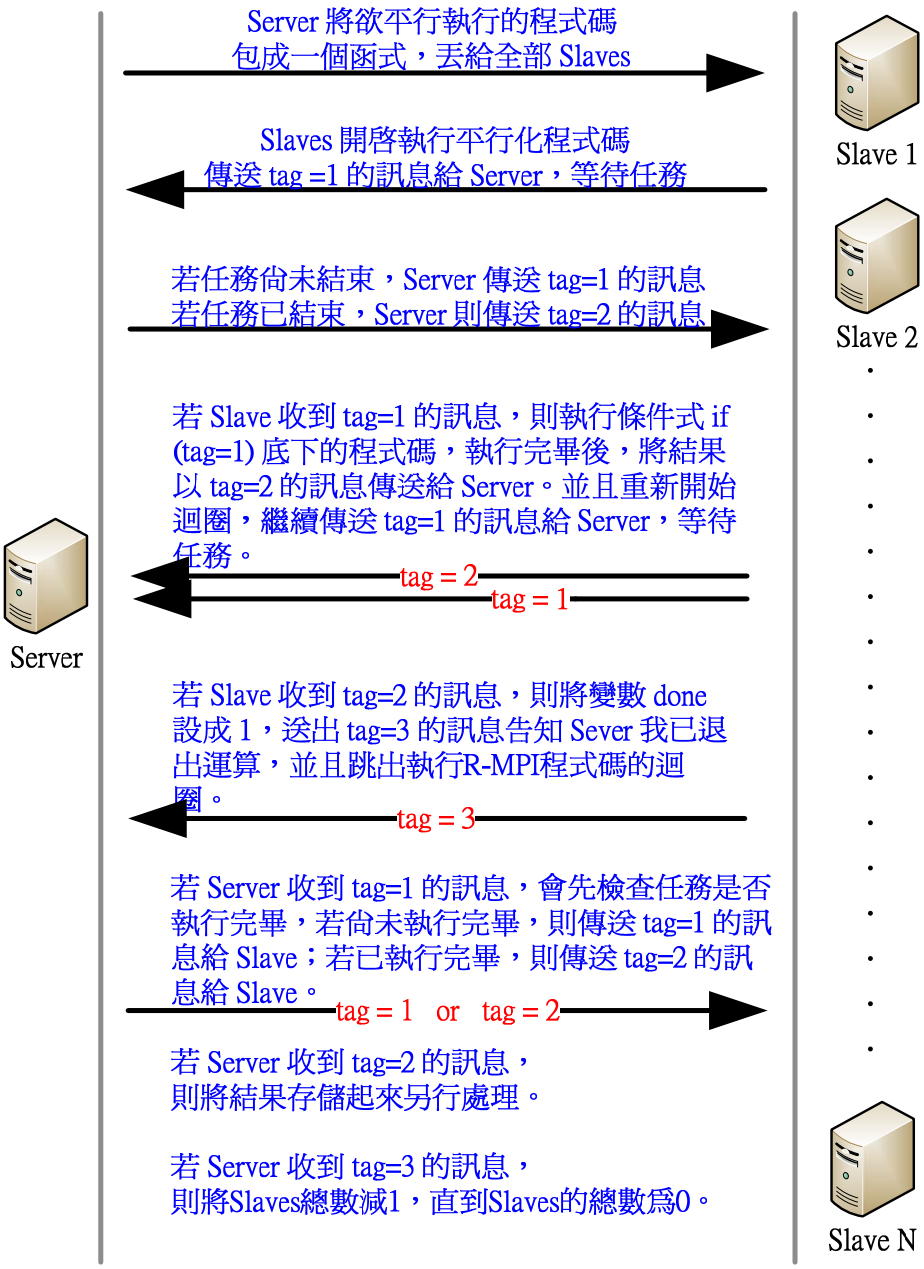


圖表 7

3. Task Pull

顧名思義，Task Pull，不像 Task Push 是 Server 主動分配任務給 slaves，最主要是因爲 Server 端並不會知道各個 slaves 的運算能力，或是各個 slaves 目前有沒有被其它行程佔用住資源。相反地，這個方式是由 slaves 主動去向 server 端要任務，因此稱爲 Task Pull。只要

slave 處理完畢時，便會告知 server 說: **我處理完畢了，若還有任務的話，請繼續 send 給我!** 這個方式最大的優點就是: 不管任務的數目是否大於 slaves 的數目；也不管各個 slaves 的運算能力是否相同，所有的任務可以在最短的時間內被處理完畢。



圖表 8

4. Task Push 與 Task Pull 的差異

我們在此舉例來說明 Task Push 與 Task Pull 的差異性。假設有 5 台機器 (Slave 1 ~ Slave 5) 要來處理總共 20 個的 Task，其中 Slave 5 的效能較差，針對此一特定任務來說，Slave 5 的處理速度要比 Slave 1

~ Slave 4 慢上 15 分鐘。以 Task Push 的方式來執行的話，20 個任務被分配的順序將會是 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5，其中因爲 Slave 5 的效能較差而多消耗的時間爲 15 x 4 = 60 分鐘。但若是以 Task Pull 的方式來執行的話，20 個任務被分配的順序將會是 1, 2, 3, 4, 5, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3，其中因爲 slave 5 的效能較差而多消耗的時間爲 15 x 1 = 15 分鐘。Task Pull 的方式足足快了 Task Push 有 45 分鐘，效能則是提升了 75%。

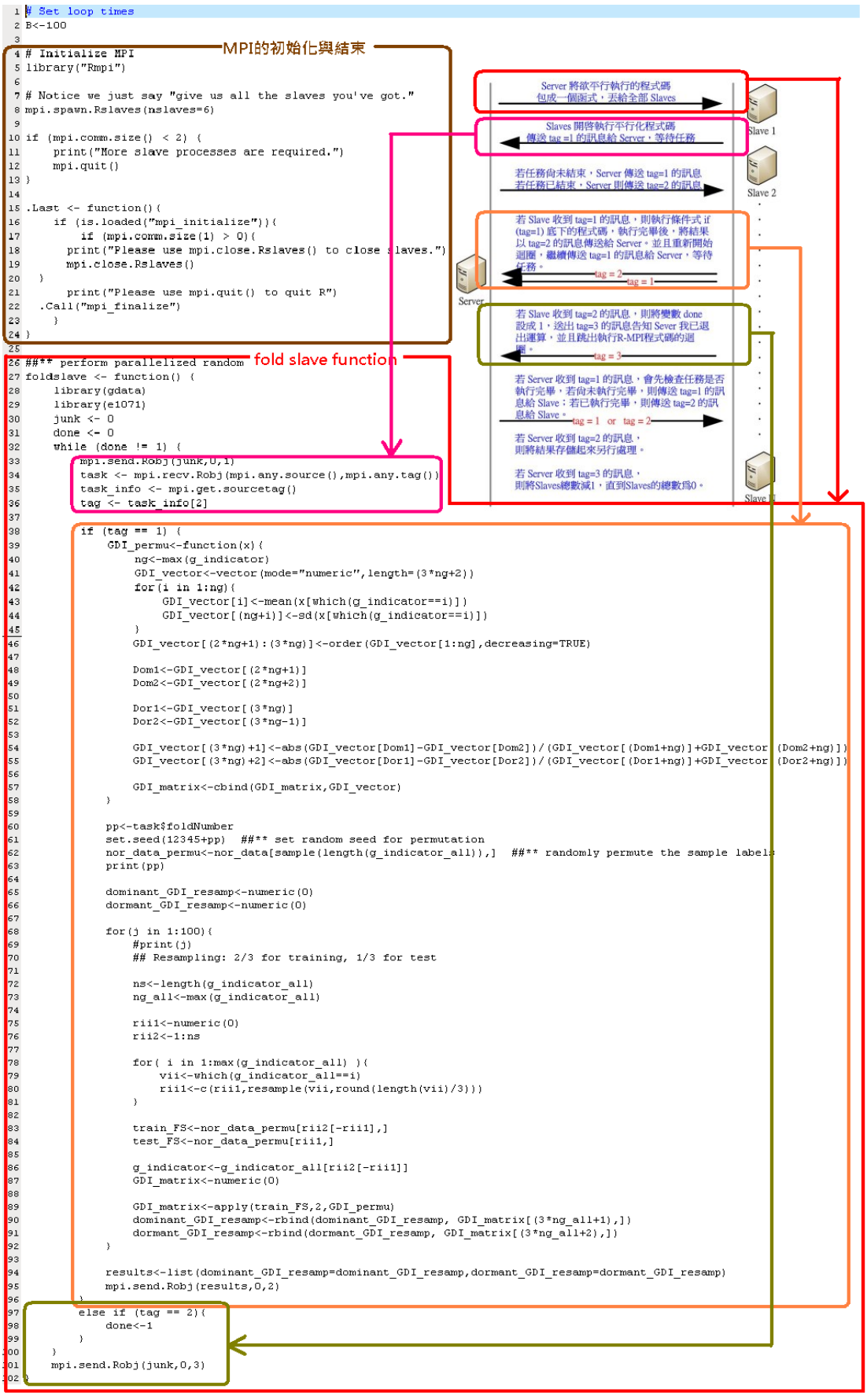
四、 範例程式說明

在這個小節中，我們將以陽明大學所提供的部分 R 程式爲範例來說明如何將一般的 R 程式改寫成 task-pull 方式的 R-MPI 程式。圖中有被彩色線條框起來的部分程式碼就是 R-MPI 的相關程式碼，每一塊彩色部分的程式碼都可以對應到 R-MPI task-pull 運作原理流程圖中的特定流程。

每個 R-MPI 程式一開始都需要先初始化 R-MPI 的相關函式，以及關閉時必須要做的檢查跟動作。因此在圖表 6 中，一開始就是 R-MPI 的初始化與結束設定。

以 Slave 的角度來解釋，最重要的就是 fold slave function 的部分，這部分的程式碼就是我們打算分散執行的程式碼，這其中又以 if (tag = 1) 底下的部分最爲重要，你可以把你想要分散運算的程式碼都寫在這個區塊裡面。同時，在這個區塊的最後，也要設定好回傳的資料，一直到 Slave 把計算的結果傳回 Server ，這樣才是完成了一次有效的運算。一旦所有任務都執行結束後， Server 與 Slave 會建立一個類似 handshake 的動作，首先，Slave 會收到 Server 所送出 tag =2 的訊息，表示任務已結束，接著，Slave 跳出 fold slave function 的 while 迴圈，並且回覆 tag =3 的訊息給 server ，告知 Server 我已成功地退出運算。最後 Server 會檢查到所有的 Slaves 都退出運算，表示工作已經全部結束。

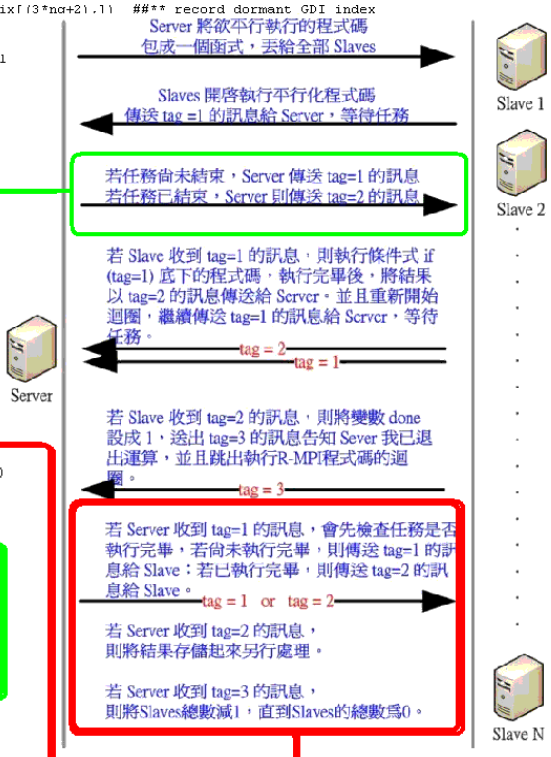
以 Server 的角度來看，Server 端要負責的事情有兩件：一是彙整所有 Slaves 的計算結果，二是負責遞交任務給 Slaves 或是告知 Slaves 任務已經執行完畢。當 Server 端收到 tag=1 的訊息時，表示 Slave 已經準備好接收新任務；當 Server 端收到 tag=2 的訊息時，表示 Slave 傳送新的計算結果過來了；當 Server 端收到 tag=3 的訊息時，表示 Slave 已經退出運算，程式碼與流程相對關係如 圖表 7 所示。




```r
# Set loop times
B<-100

# Initialize MPI
library("Rmpi")

# Notice we just say "give us all the slaves you've got."
mpi.spawn.Rslaves(nslaves=6)

if (mpi.comm.size() < 2) {
    print("More slave processes are required.")
    mpi.quit()
}

.Last <- function(){
    if (is.loaded("mpi_initialize")){
        if (mpi.comm.size(1) > 0){
      print("Please use mpi.close.Rslaves() to close slaves.")
      mpi.close.Rslaves()
  }
      print("Please use mpi.quit() to quit R")
  .Call("mpi_finalize")
    }
}

### perform parallelized random
foldslave <- function() {
    library(gdata)
    library(e1071)
    junk <- 0
    done <- 0
    while (done != 1) {
        mpi.send.Robj(junk,0,1)
        task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())
        task_info <- mpi.get.sourcetag()
        tag <- task_info[2]

        if (tag == 1) {
            GDI_permu<-function(x){
                ng<-max(g_indicator)
                GDI_vector<-vector(mode="numeric",length=(3*ng+2))
                for(i in 1:ng){
                    GDI_vector[i]<-mean(x[which(g_indicator==i)])
                    GDI_vector[(ng+i)]<-sd(x[which(g_indicator==i)])
                }
                GDI_vector[(2*ng+1):(3*ng)]<-order(GDI_vector[1:ng],decreasing=TRUE)

                Dom1<-GDI_vector[(2*ng+1)]
                Dom2<-GDI_vector[(2*ng+2)]

                Dor1<-GDI_vector[(3*ng)]
                Dor2<-GDI_vector[(3*ng-1)]

                GDI_vector[(3*ng)+1]<-abs(GDI_vector[Dom1]-GDI_vector[Dom2])/(GDI_vector[(Dom1+ng)]+GDI_vector[(Dom2+ng)])
                GDI_vector[(3*ng)+2]<-abs(GDI_vector[Dor1]-GDI_vector[Dor2])/(GDI_vector[(Dor1+ng)]+GDI_vector[(Dor2+ng)])

                GDI_matrix<-cbind(GDI_matrix,GDI_vector)
            }

            pp<-task$foldNumber
            set.seed(12345+pp)  ##** set random seed for permutation
            nor_data_permu<-nor_data[sample(length(g_indicator_all)),]  ##** randomly permute the sample labels
            print(pp)

            dominant_GDI_resamp<-numeric(0)
            dormant_GDI_resamp<-numeric(0)

            for(j in 1:100){
                #print(j)
                ## Resampling: 2/3 for training, 1/3 for test

                ns<-length(g_indicator_all)
                ng_all<-max(g_indicator_all)

                rii1<-numeric(0)
                rii2<-1:ns

                for( i in 1:max(g_indicator_all) ){
                    vii<-which(g_indicator_all==i)
                    rii1<-c(rii1,resample(vii,round(length(vii)/3)))
                }

                train_FS<-nor_data_permu[rii2[-rii1],]
                test_FS<-nor_data_permu[rii1,]

                g_indicator<-g_indicator_all[rii2[-rii1]]
                GDI_matrix<-numeric(0)

                GDI_matrix<-apply(train_FS,2,GDI_permu)
                dominant_GDI_resamp<-rbind(dominant_GDI_resamp, GDI_matrix[(3*ng_all+1),])
                dormant_GDI_resamp<-rbind(dormant_GDI_resamp, GDI_matrix[(3*ng_all+2),])
            }

            results<-list(dominant_GDI_resamp=dominant_GDI_resamp,dormant_GDI_resamp=dormant_GDI_resamp)
            mpi.send.Robj(results,0,2)
        }
        else if (tag == 2){
            done<-1
        }
    }
    mpi.send.Robj(junk,0,3)
}
```

圖表 9

```
for(j in 1:100){
    print(j)

    ## Resampling: 2/3 for training, 1/3 for test

    ns<-length(g_indicator_all)  # Number of samples
    ng<-max(g_indicator_all)     # Number of groups

    rii1<-numeric(0)
    rii2<-1:ns
    for( i in 1:max(g_indicator_all) ){
        vii<-which(g_indicator_all==i)
        rii1<-c(rii1,resample(vii,round(length(vii)/3)))
    }

    train_FS<-nor_data[rii2[-rii1],]   # training set for Feature-Selection
    test_FS<-nor_data[rii1,]           # test set for Feature-Selection

    g_indicator<-g_indicator_all[rii2[-rii1]]   # Necessary Input file for GDI function
    GDI_matrix<-numeric(0)                      # Necessary Input file for GDI function

    GDI_matrix<-apply(train_FS,2,GDI)           # Apply the GDI function by column (gene)
    dominant_GDI_resamp<-rbind(dominant_GDI_resamp, GDI_matrix[(3*ng+1),])  ##** record dominant GDI index
    dormant_GDI_resamp<-rbind(dormant_GDI_resamp, GDI_matrix[(3*ng+2),])  ##** record dormant GDI index
}

dominant_GDI<-apply(dominant_GDI_resamp, 2, mean)  ##** cal
dormant_GDI<-apply(dormant_GDI_resamp, 2, mean)

mpi.bcast.Robj2slave(g_indicator_all)
mpi.bcast.Robj2slave(nor_data)
mpi.bcast.Robj2slave(foldslave)

mpi.bcast.cmd(foldslave())

tasks <- vector('list')
for (i in 1:B) {
    tasks[[i]] <- list(foldNumber=i)
}

junk <- 0
closed_slaves <- 0
n_slaves <- mpi.comm.size()-1

dominant_GDI_permu<-numeric(0)
dormant_GDI_permu<-numeric(0)

# Create Start Timer
start_time<-Sys.time()

while (closed_slaves < n_slaves) {
    message <- mpi.recv.Robj(mpi.any.source(), mpi.any.tag()
    message_info <- mpi.get.sourcetag()
    slave_id <- message_info[1]
    tag <- message_info[2]

    if (tag == 1) {
        if (length(tasks) > 0) {
            mpi.send.Robj(tasks[[1]], slave_id, 1);
            tasks[[1]] <- NULL
        }
        else {
            mpi.send.Robj(junk, slave_id, 2)
        }
    }

    else if (tag == 2) {
        dominant<-message$dominant_GDI_resamp
        dormant<-message$dormant_GDI_resamp
        dominant_GDI_permu<-rbind(dominant_GDI_permu, apply(dominant, 2, mean))
        dormant_GDI_permu<-rbind(dormant_GDI_permu, apply(dormant, 2, mean))
    }

    else if (tag == 3){
        closed_slaves <- closed_slaves + 1
    }
}

end_time<-Sys.time()
runTime<-difftime(end_time,start_time,units="secs")
write(runTime,file="runTime/parallel_permu_runTime",append=TRUE)

######## assess p-value and q-value from permutation test ######
pvalue.dominant<-rep(NA, length(dominant_GDI))
pvalue.dormant<-rep(NA, length(dominant_GDI))
qvalue.dominant<-rep(NA, length(dominant_GDI))
qvalue.dormant<-rep(NA, length(dominant_GDI))
G<-ncol(dominant_GDI_permu)
B<-nrow(dominant_GDI_permu)

for(i in 1:length(dominant_GDI)){
    pvalue.dominant[i]<-length(which(dominant_GDI_permu>=dominant_GDI[i]))/(G*B)
    qvalue.dominant[i]<-min(length(which(dominant_GDI_permu>=dominant_GDI[i]))/(length(which(dominant_GDI>=dominant_GDI[i]))*B),1
}

for(i in 1:length(dormant_GDI)){
    pvalue.dormant[i]<-length(which(dormant_GDI_permu>=dormant_GDI[i]))/(G*B)
    qvalue.dormant[i]<-min(length(which(dormant_GDI_permu>=dormant_GDI[i]))/(length(which(dormant_GDI>=dormant_GDI[i]))*B),1)
}

write.table(t(rbind(dominant_GDI, pvalue.dominant, qvalue.dominant, dominant_GDI_permu)), paste(directory, "obsGDI_pvalue_qvalue_

write.table(t(rbind(dormant_GDI, pvalue.dormant, qvalue.dormant, dormant_GDI_permu)), paste(directory, "obsGDI_pvalue_qvalue_perm

mpi.close.Rslaves()
mpi.quit(save="no")
```



圖表 10

五、 結論

建置一個具有 R 與 R-MPI 的平台，可以說是相當方便，一旦具備了 R 與 R-MPI 的基本念概，在了解方法之後，要將一般的 R 程式平行化也並不困難，一般 R 程式中，最花費時間的就是在於處理大量迴圈的部分，因此 R-MPI 的重點就在於拆解迴圈，只要把迴圈的次數平均分散給底下的 Slaves 機器做處理，就可以大大地節省時間成本。

Task Pull 的程式設計法，在分散式運算中可以取得很好的效能與結果，使在任何時間中，所有機器的使用率接近 100%，這也是我們所希望看到的結果。R-MPI 可以讓所有的機器在任何時間中都有任務可做，並且讓機器閒置的時間縮到最短。

六、 參考文獻

[1] Extensions to the Message-Passing Interface, Message Passing Interface Forum, November 15, 2003

[2] MPI 程式撰寫與實作, 周守成、周朝宜, November 04, 2006

[3] http://www.mcs.anl.gov/research/projects/mpich2/

[4] http://www.r-project.org/

[5] http://ace.acadiau.ca/math/ACMMaC/Rmpi/basics.html

[6] http://www.stats.uwo.ca/faculty/yu/Rmpi/knowing_issues.htm

[7] http://cran.cs.pu.edu.tw/banner.shtml

[8] The Rmpi Package, Hao Yu, April 21, 2004