Fortran 語言

MPI 平行計算程式設計

編　著: 鄭 守 成

期　間: 民國 92 年 2 月 12 日

電　話: (03) 5776085 x 305

E-mail : c00tch00@nchc.gov.tw

第一章　前言

本章將介紹 MPI 平行計算軟體、國家高速網路與計算中心現有的平行計算環境、以及在各型機器上如何使用 MPI。

第一節簡單介紹 MPI 平行計算軟體。

第二節介紹國家高速網路與計算中心現有的平行計算環境。

第三節介紹如何在 IBM 電腦系統上使用 MPI，包括路徑的設定、平行程式的編譯、及平行程式的執行等。

第四節介紹如何在 PC cluster 上使用 MPI，包括路徑的設定、平行程式的編譯、及平行程式的執行等。

1.1 MPI 平行計算軟體

MPI (Message Passing Interface) 是第一個標準化的 Message Passing 平行語言。可以使用在 Fortran、C、C++ 等語言撰寫的程式上。MPI 平行程式可以在分散式記憶體平行系統上執行，也可以在共用記憶體平行系統上執行。目前系統廠商所提供的 MPI 軟體是屬於 MPI1.2 版。它提供了一百多個副程式，讓程式人員來選用。MPI 協會在 1998 年公布了 MPI 2.0 版的規格，數年之後就會有 MPI 2.0 版的軟體可用了。日前美國的 Argonne National Lab 已經公布了 MPICH 1.2.0 版的整套軟體，該版含有 MPI 2.0 版的部份功能。有興趣的讀者可以免費自網路下載該軟體，其網址是

http://www-unix.mcs.anl.gov/mpi/mpich

也可以用 anonymous ftp 下載該軟體，其網址是

ftp.mcs.anl.gov

其下目錄 (directory) pub/mpi 裏檔名 mpich-1.2.0.tar.Z 或 mpich-1.2.0.tar.gz，在該目錄之下還有許多與 MPI 相關的資訊可供參考。

1.2 國家高速網路與計算中心的平行計算環境

目前國家高速網路與計算中心的 IBM SP2、IBM SP2 SMP、IBM p690、HP SPP2200、SGI Origin2000 和 Fujitsu VPP300 等系統上均有該公司自備的 MPI 平行軟體，PC cluster 上是裝用 MPICH 公用平行軟體，也都有能力執行平行程式。但是到目前為止，只有 PC cluster、IBM SP2、IBM SP2 SMP 和 IBM p690 設有一個 CPU 只執行一個程式的平行環境，其他機器上則無此種設定。例如，若有一個用戶要用四個 CPU 來執行其平行程式，他在 IBM SP2 上取得四個 CPU 之後，這四個 CPU 就僅只執行這個平行程式直到它執行完畢為止，不會有其他程式進來跟他搶 CPU 時間。但他在其他機器 (如 HP SPP2000) 上取得四個 CPU 之後，如果所有使用者對 CPU 的需求數量超過該系統的 CPU 總數時，他所取得四個 CPU 之中的每一個 CPU，都有可能要跟其他程式以分時方式 (time sharing) 共用一個 CPU。

HP SPP2000 和 SGI ORIGIN2000 為共用記憶體平行系統，這種電腦系統是 16 顆 CPU 共用一組記憶體。SP2 和 VPP300 是屬於分散式記憶體平行系統，每一個 CPU 備有它獨用的記憶體。IBM SP2 SMP 及 IBM p690 是共用記憶體及分散式記憶體混合的平行系統，SP2 SMP 每一個 node 備有 4 顆 CPU 共用一組記憶體，目前備有 42 個 node 的 SMP cluster。p690 每一個 node 備有 32 顆 CPU 共用一組記憶體，目前備有 8 個 node 的 SMP cluster。SP2、SP2 SMP 和 p690 是採用該系統專屬的工作排程軟體 (job scheduler) LoadLeveler 來安排用戶的批次工作(batch job)。使用者必須備妥 LoadLeveler 的 job command file，使用 llsubmit 指令把該批次工作交給該系統來執行。SPP2000、ORIGIN2000 和 VPP300 是採用 NQS (Network Queue

System) 工作排程軟體來安排用戶的批次工作。使用者必須備妥 NQS 的 job command file，使用 qsub 指令把該批次工作交給各該系統來執行。PC cluster 是採用 DQS (Distributed Queue System) 工作排程軟體來安排用戶的批次工作，其使用方式類似 NQS。

1.3 在 IBM 電腦系統上如何使用 MPI

首先，C shell 用戶要在自己 home directory 的.cshrc 檔裏加入下列路徑，這樣才能夠抓得到 include file (mpif.h、mpif90.h、mpi.h)、編譯指令 (mpxlf、mpxlf90、mpcc)、MPI library、和 LoadLeveler 指令 (llsubmit、llq、llstatus、llcancel)。

```
set lpath=(. ~ /usr/lpp/ppe.poe/include /usr/lpp/ppe.poe/lib)
set lpath=($lpath /usr/lpp/ppe.poe/bin /home/loadl/bin )
set path=($path $lpath)
```

加好上述路徑之後，將.cshrc 存檔，再執行 source .cshrc 指令，即可進行平行程式的編譯與執行。簽退 (logout) 後再簽到 (login) 之後就不必再執行 source .cshrc 指令。

1.3.1 IBM 電腦系統的 MPI Fortran 程式編譯指令

使用 MPI 的 Fortran 77 平行程式，其編譯器 (compiler) 一般叫做 mpif77，但是在 IBM 電腦系統上卻叫做 mpxlf。mpxlf 常用的編譯選項如下 :

```
mpxlf -O3 -qarch=auto -qstrict -o file.x file.f
```

其中選項 -O3 是作最高級的最佳化 (level 3 Optimization)，可使程式的計算速度加快數倍

-qarch=auto 是通知編譯器該程式要在同型機器上執行

-qstrict 是通知編譯器不要改變計算的順序

-o file.x　　是指定執行檔名為 file.x，

不指定時其內定 (default) 檔名為 a.out

1.3.2 IBM 電腦系統的 Job command file

要在 IBM SP2(ivy)上執行平行程式，使用者必須備妥 LoadLeveler 的 job command file。例如，下面這個 job command file 叫做 jobp4，它要在四個 CPU 上執行平行程式 file.x。

```
#!/bin/csh
#@ executable = /usr/bin/poe
#@ arguments = /your_working_directory/file.x    –euilib us
#@ output       = outp4
#@ error        = outp4
#@ job_type     = parallel
#@ class        = medium
#@ min_processors = 4
#@ max_processors = 4
#@ requirements = (Adapter == "hps_user")
#@ wall_clock_limit = 20
#@ queue
```

其中 executable = /usr/bin/poe 是固定不變，poe 是指 Parallel Operating Environment

arguments = 執行檔所在之全路徑及檔名

output = 標準輸出檔名 (stdout)

error = 錯誤訊息 (error message) 輸出檔名

class = SP2 CPU 的分組別，使用 llclass 指令可以看到分組別 ：

short (CPU 時間上限為 12 小時，共有 10 顆 120MHz CPU)

medium (CPU 時間上限為 24 小時，共有 64 顆 160MHz CPU)

long (CPU 時間上限為 96 小時，共有 24 顆 120MHz CPU)

min_processors = 最少的 CPU 數目

max_processors = 最多的 CPU 數目

requirements = (Adapter == "hps_user") 是固定不變

wall_clock_limit = 該 job 最多需要的時間，單位為分鐘

queue 是固定不變

平行計算可以使用的 CPU 數目，short class 最多 4 個 CPU，medium class 最多 32 個 CPU，long class 最多 8 個 CPU。由於 MPI 1.2 版不具備取得 CPU、控制 CPU、和歸還 CPU 的功能，所以 min_processors 和 max_processors 要填相同的數字。

要在 IBM SP2 SMP (ivory)上執行平行程式，使用者必須備妥 LoadLeveler 的 job command file。例如，下面這個 job command file 叫做 jobp4，它要在四個 CPU 上執行平行程式 file.x。

```
#!/bin/csh
#@ network.mpi= css0,shared,us
#@ executable = /usr/bin/poe
#@ arguments = /your_working_directory/file.x  –euilib us
```

```
#@ output       = outp4
#@ error        = outp4
#@ job_type     = parallel
#@ class        = medium
#@ tasks_per_node = 4
#@ node = 1
#@ queue
```

由於 IBM SP2 SMP 每個 Node 含有四棵 375MHz CPU 共用 4GB 或 8GB 的記憶體。

class = SP2 SMP CPU 的分組別，使用 llclass 指令可以看到分組別：

short (CPU 時間上限為 12 小時， 3 個 Node 共有 6 顆 CPU)

medium (CPU 時間上限為 24 小時，32 個 Node 共有 128 顆 CPU)

bigmem (CPU 時間上限為 48 小時， 4 個 Node 共有 16 顆 CPU)

這個 class 一個 Node 備有 8GB 的共用記憶體

tasks_per_node=4 是說明一個 Node 選用四棵 CPU

node=1 是說明要用一個 Node，一共四棵 CPU

平行計算可以使用的 CPU 數目 medium class 是 8 個 Node 一共 32 顆 CPU。

要在 IBM p690 上執行平行程式，使用者必須備妥 LoadLeveler 的 job command file。例如，下面這個 job command file 叫做 jobp8，它要在 8 個 CPU 上執行平行程式 file.x。

```
#!/bin/csh
#@ executable = /usr/bin/poe
#@ network.mpi= csss,shared,us
#@ arguments=/your-working-directory/file.x
#@ output      = outp8
```

```
#@ error       = outp8
#@ job_type = parallel
#@ class       = 8cpu
#@ tasks_per_node = 8
#@ node = 1
#@ queue
```

由於 IBM p690 每個 Node 含有 32 棵 1.3 GHz CPU 共用 128GB 的記憶體。

class = p690 CPU 的分組別，使用 llclass 指令可以看到分組別 ：

32cpu	(CPU 時間上限為	36 小時， 2 個 Node 共有 64 顆 CPU)
16cpu	(CPU 時間上限為	72 小時， 1 個 Node 共有 32 顆 CPU)
8cpu	(CPU 時間上限為	72 小時， 6 個 Node 共有 188 顆 CPU)
4cpu	(CPU 時間上限為	96 小時， 6 個 Node 共有 188 顆 CPU)
serial	(CPU 時間上限為	168 小時， 1 個 Node 共有 32 顆 CPU)

tasks_per_node=8 是說明一個 Node 選用 8 棵 CPU

node=1 是說明要用一個 Node，一共 8 棵 CPU

1.3.3 IBM 電腦系統的平行程式的執行指令

要在 IBM 電腦系統上執行平行程式，使用者在備妥 LoadLeveler 的 job command file 之後，就可以使用 llsubmit 指令將該 job command file 交給該系統排隊等候執行。例如上一節的 job command file 例子 jobp4 即可用下述指令交付執行 ：

```
llsubmit    jobp4
```

工作交付之後，該工作執行的情形可用 llq 指令查詢。要縮小查詢的範圍可在 llq 指令之後加上 grep 指令敘明要查詢的 class 或 user id。例如上一個例子 jobp4 所選用的分組別為 medium，就可用下述指令進行查詢 ：

llq | grep medium

llq 顯示之內容有下列事項 ：

job_id	user_id	submitted	status	priority	class	running on
ivy1.1781.0	u43ycc00	8/13 11:24	R	50	medium	ivy39
ivy1.1814.0	u50pao00	8/13 20:12	R	50	short	ivy35

其中 job_id 是 LoadLeveler 給交付的工作編定的工作代號

user_id 是使用者的 login name

submitted 是交付工作的時刻，月/日 時:分

status 是工作執行的情形

R 表 Running

I 表 Idle (=waiting in queue)

ST 表 Start execution

NQ 表 Not Queued，還在隊伍之外

Priority 是交付工作的優先次序，不用更動它

Class 是 CPU 分組別

Running on 是執行交付工作的第一個 CPU 代號

工作交付執行之後，如果要中止該工作的執行可用 llcancel 指令殺掉該工作。

llcancel　job_id

此處的 job_id 就是使用 llq 指令所顯示之使用者交付工作的工作代號。 執行過 llcancel 指令之後，再使用 llq 指令就可以看出該工作已經消失不見了。

1.4　在 PC Cluster 上如何使用 MPI

首先，使用 MPICH 的 C shell 用戶要在自己 home directory 的.cshrc 檔裏加入下列路徑，這樣才能夠抓得到 include file (mpif.h、mpi.h)、編譯指令 (mpif77、mpicc)、MPI library、和 DQS 指令。不同的 PC Cluster 這些存放的路徑可能不同，要向該系統的管理人詢問。其路徑設定如下 ：

```
setenv PGI /usr/local/pgi
set path = ( . ~ /usr/local/pgi/linux86/bin $path)
set path = ( /package/DQS _hpcserv2/bin $path)
set path = ( /package/mpich _hpcserv2/bin $path)
```

其中第一行是 PGI 公司 (Portland Group Inc.) 軟體存放的路徑，第二行是 PGI 公司 Fortran77 編譯器 pgf77 存放的路徑，第三行是 DQS 批次工作排程軟体存放的路徑，第四行是 MPICH 編譯系統存放的路徑。沒有購用 PGI 公司的軟體時前面兩行可以省略。

1.4.1　PC Cluster 上的 MPI　Fortran 程式編譯指令

MPICH 的 Fortran77 平行程式編譯器叫做 mpif77，其底層是使用 GNU 的 g77 來編譯，因此可以使用 g77 的調適選項。舉例如下 ：

```
mpif77 -O3 -o file.x file.f
```

其中選項

-O3　　是選用 g77 最高層次的調適選項

-o file.x　是指定編譯產生的執行檔為 file.x

　　沒有指定時，內定的執行檔為 a.out

file.f　　是 Fortran77 平行程式

如果選用 PGI 公司的 MPI 平行程式編譯器 mpif77，其底層是使用該公司的 pgf77 來編譯，因此可以使用 pgf77 及 pgf90 的調適選項。其 makefile 舉例如下 ：

```
OBJ      = file.o
EXE      = file.x
MPI      = /home/package/mpich_PGI
LIB      = $(MPI)/lib/LINUX/ch_p4
LFLAG = -L$(LIB) -lfmpich
MPIF77 = $(MPI)/bin/mpif77
OPT     = -O2 -I$(MPI)/include
$(EXE) : $(OBJ)
          $(MPIF77)   $(LFLAG) -o $(EXE) $(OBJ) $(LIB)
.f.o :
          $(MPIF77)   $(OPT) -c $<
```

備妥 makefile 之後，只要下 make 指令就會開始程式的編譯工作。

1.4.2 PC Cluster 上的 Job command file

如果該 PC cluster 是採用 DQS 排程軟體來安排批次工作時，要在其上執行平行程式，使用者必須備妥 DQS 的 job command file。例如，下面這個 job command file 叫做 jobp4，它要在四個 CPU 上執行平行程式 hubksp :

```
#!/bin/csh
#$ -l qty.eq.4
#$ -N HUP4
#$ -A user_id
#$ -cwd
#$ -j y
cat $HOSTS_FILE > MPI_HOST
mpirun -np 4 -machinefile MPI_HOST hubksp >& outp4
```

其中 #!/bin/csh 是說明這是個 C shell script

#$ -l qty.eq.4 是向 DQS 要求四個 CPU，qty 是數量(quantity)

#$ -N HUP4 是說明這個工作的名字(Name)叫做 HUP4

#$ -A user_id 是說明付費帳號(Account)就是使用者帳號

#$ -cwd 是說明要在現在這個路徑(working directory)上執行程式

內定的路徑是 home directory

#$ -j y 是說明錯誤訊息要輸出到標準輸出檔

$HOST_FILE 是 DQS 安排給這項工作的 node list

-np 4 hubksp 是告訴 mpirun 要在四個 CPU 上執行平行程式 hubksp

>& outp4 是要把標準輸出檔寫入 outp4

1.4.3 PC Cluster 上的平行程式執行指令

要在 PC cluster 上執行平行程式，使用者在備妥 DQS 的 job command file 之後，就可以使用 qsub32 指令將該 job command file 交給 PC cluster 排隊等候執行。例如上一節的 job command file 例子 jobp4 即可用下述指令交付執行 ：

qsub32　jobp4

工作交付之後，可以使用 qstat32 指令 (不加參數) 查詢整個 cluster 交付工作執行的情形，使用 qstat32 -f 指令查詢整個 cluster 各個 node 的狀況。上述指令 qsub32 jobp4 之後使用 qstat32 指令顯示的內容如下 :

```
c00tch00 HUP4      hpcs001        62     0:1     r          RUNNING      02/26/99 10:51:23
c00tch00 HUP4      hpcs002        62     0:1     r          RUNNING      02/26/99 10:51:23
c00tch00 HUP4      hpcs003        62     0:1     r          RUNNING      02/26/99 10:51:23
c00tch00 HUP4      hpcs004        62     0:1     r          RUNNING      02/26/99 10:51:23

----Pending Jobs ----------------------------------------------------------------------------------
c00tch00 RAD5                     70     0:2              QUEUED       02/26/99 19:24:32
```

第一欄是 user_id，第二欄是交付工作的名稱，第三欄是 CPU 代號，第四欄是 DQS 替交付的工作編定的工作編號 job_id (62)，第五欄 0:1 的 0 是交付工作的優先序號， 0:1 的 1 是該用戶交付的第一個工作，第六欄的 r 和第七欄的 RUNNING 表示該工作正在執行中，最後是該工作交付時的時刻，月/日/年 時:分:秒。排隊等待執行的工作則出現在 Pending Jobs 之列，對應 RUNNING 的欄位則為 QUEUED。

工作交付執行之後，如果要中止該工作的執行可用 qdel32 指令殺掉該工作。

Qdel32　job_id

此處的 job_id 就是使用 qstat32 指令所顯示之第四欄。執行過 qdel32 指令之後，再使用 qstat32 指令就可以看出該工作已經消失不見了。

第二章　無邊界資料交換的平行程式

最簡單的平行程式就是無邊界資料交換的平行程式。本章將利用一個很簡單的循序程式 (sequential program) 使用 MPI 指令加以平行化，並比較其計算結果以資驗證。

2.1 節介紹六個 MPI 基本指令 MPI_INIT、MPI_FINALIZE、MPI_COMM_SIZE、MPI_COMM_RANK、MPI_SEND、MPI_RECV。

2.2 節介紹無邊界資料交換的循序程式 T2SEQ。

2.3 節說明使用這六個 MPI 基本指令平行化循序程式 T2SEQ 而成為平行程式 T2CP。

2.4 節介紹另外四個常用的 MPI 指令 MPI_SCATTER、MPI_GATHER、MPI_REDUCE、MPI_ALLREDUCE。

2.5 節是使用這些指令平行化循序程式 T2SEQ 而成為平行程式 T2DCP。

2.1 MPI 基本指令

MPI 的基本指令有下列六個，將於本節分段加以介紹。

MPI_INIT, MPI_FINALIZE,
MPI_COMM_SIZE, MPI_COMM_RANK,
MPI_SEND, MPI_RECV

2.1.1 mpif.h include file

使用 MPI 撰寫 Fortran 平行程式時，必須在每一個程式 (包括主程式和副程式) 的宣告段落裏加上 INCLUDE 'mpif.h'陳述 (statement)。mpif.h 檔案裏含有編譯 MPI 平行程式所必須的 MPI 字彙與 MPI 常數 (constant)。例如：

```
PROGRAM DRIVER
IMPLICIT REAL*8 . . .
INCLUDE    'mpif.h'
. . .
CALL CHEF( . . . )
. . .
STOP
END

SUBROUTINE CHEF( . . . )
IMPLICIT REAL*8 . . .
INCLUDE    'mpif.h'
. . . . . .
RETURN
END
```

讀者可以在 MPI 軟體所在之路徑裏查看 mpif.h 的內容。不同廠商設定的 MPI 常數也許不盡相同，但是所使用的 MPI 字彙則是完全一致。

2.1.2 MPI_INIT, MPI_FINALIZE

在叫用 (CALL) 其他 MPI 函數或副程式之前必須先叫用 MPI_INIT 副程式，來啟動該程式在多個 CPU 上的平行計算工作。在程式結束 (STOP) 之前必須叫用 MPI_FINALIZE 副程式，以結束平行計算工作。所以 MPI_INIT 和 MPI_FINALIZE 在主程式裏只要叫用一次就夠了，例如：

```
PROGRAM T2CP
PARAMETER ( . . )
INCLUDE    'mpif.h'
REAL*8      . . .
INTEGER     . . .
CALL MPI_INIT(IERR)
. . .
CALL MPI_FINALIZE(IERR)
STOP
END
```

所有 MPI 副程式引數 (argument) 的資料類別除了資料名稱之外，其餘的都是整數 (integer)，傳回的引數 IERR 其值為零時是正常結束，否則就有錯誤發生。

2.1.3 MPI_COMM_SIZE, MPI_COMM_RANK

通常在叫用過 MPI_INIT 之後，就必須叫用 MPI_COMM_SIZE 以得知參與平行計算的 CPU 個數 (NPROC)，及叫用 MPI_COMM_RANK 以得知我是第幾個 CPU (MYID)，第幾個 CPU 是從 0 開始起算。所以第一個 CPU 的 MYID 值為零，第二個 CPU 的 MYID 值為 1，第三個 CPU 的 MYID 值為 2，餘類推。通常要在幾個 CPU 上作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裏事先設定要在幾個 CPU 上作平行計

算，其意義只供程式人員做參考，實際上使用幾個 CPU 作平行計算是根據 job command file 裏 min_processors 和 max_processors 的設定值，或 -np 的設定值。

MPI_COMM_SIZE 和 MPI_COMM_RANK 的叫用格式如下:

```
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
```

引數 MPI_COMM_WORLD 是 MPI 內定的 (default) communicator，參與該程式平行計算的全部 CPU 都是屬於同一個 communicator。 屬於同一個 communicator 的各個 CPU 之間才可以傳送資料。MPI 1.2 版不具備 CPU 的取得與控制功能，參與平行計算的 CPU 顆數從程式開始執行到程式結束都是固定不變的。因此，這兩個 MPI 副程式在一個程式裏只要叫用一次就可以了。例如 :

```
PROGRAM T2CP
PARAMETER ( . . )
INCLUDE    'mpif.h'
REAL*8      . . .
INTEGER    NPROC, MYID
CALL MPI_INIT (IERR)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
. . .
CALL MPI_FINALIZE (IERR)
STOP
END
```

2.1.4 MPI_SEND, MPI_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)，另外一種叫做 '集體通訊' (collective communication)。此處先介紹 '點對點通訊' 類的 MPI_SEND 和 MPI_RECV, 其他常用的 '點對點通訊' 及 '集體通訊' 指令容後再介紹。

一個 CPU 與另外一個 CPU 之間的資料傳送屬於 '點對點通訊'，送出資料的 CPU 要叫用 MPI_SEND 來送資料，而收受資料的 CPU 要叫用 MPI_RECV 來收資料。一個 MPI_SEND 必須要有一個對應的 MPI_RECV 與之配合，才能完成一份資料的傳送工作。MPI_SEND 的叫用格式如下:

```
CALL MPI_SEND (DATA, ICOUNT, DATA_TYPE, IDEST, ITAG,
&                MPI_COMM_WORLD, IERR)
```

引數 DATA 要送出去的資料起點，可以是純量 (scalar) 或陣列 (array)資料

ICOUNT 要送出去的資料數量，當 ICOUNT 的值大於一時，DATA 必須是陣列

DATA_TYPE 是要送出去的資料類別，MPI 內定的資料類別如表 1.1

IDEST 是收受資料的 CPU id

ITAG 要送出去的資料標籤

MPI data types	Fortran data types
MPI_CHARACTER	CHARACTER
MPI_LOGICAL	LOGICAL
MPI_INTEGER	INTEGER
MPI_REAL, MPI_REAL4	REAL, REAL*4
MPI_REAL8, MPI_DOUBLE_PRECISION	REAL*8, DOUBLE PRECISION
MPI_COMPLEX, MPI_COMPLEX8	COMPLEX, COMPLEX*8
MPI_COMPLEX16, MPI_DOUBLE_COMPLEX	COMPLEX*16

表 1.1 MPI data types

MPI_RECV 的叫用格式如下:

```
      CALL MPI_RECV (DATA, ICOUNT, DATA_TYPE, ISRC, ITAG,
     &                MPI_COMM_WORLD, ISTATUS, IERR)
```

引數 DATA 是要收受的資料起點

ICOUNT 是要收受的資料數量

DATA_TYPE 是要收受的資料類別

ISRC 是送出資料的 CPU id

ITAG 是要收受的資料標籤

ISTATUS 是執行 MPI_RECV 副程式之後的狀況

ISTATUS 為一整數陣列，該陣列的長度為在 mpif.h 裏已經設定的常數 MPI_STATUS_SIZE，寫法如下

```
      INTEGER ISTATUS(MPI_STATUS_SIZE)
```

一個 CPU 同時要收受多個 CPU 送來的資料時，若不依照特定的順序，而是先到先收，則其指令為

```
      CALL MPI_RECV( BUFF, ICOUNT, DATA_TYPE, MPI_ANY_SOURCE, ITAG,
     1               MPI_COMM_WORLD, ISTATUS, IERR)
```

若要判別送出該資料的 CPU id 時就要用到 ISTATUS 變數如下

ISRC = ISTATUS(MPI_SOURCE)

MPI 在傳送資料 (MPI_SEND、MPI_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. communicator

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。

2.2 無邊界資料交換的循序程式 T2SEQ

T2SEQ 是個無邊界資料交換的循序程式，在 test data generation 段落裏設定陣列 B、C、D 的值，然後把這些陣列寫到磁檔上。其目的是便利往後的範例程式可以讀入同一組資料作平行計算，用來驗證其計算的結果是否正確。

這個程式的計算部份只有一個 DO loop，而且該 loop 裏只有兩個計算陳述，其目的是方便往後說明如何將這一類 DO loop 平行化。實際的計算程式也許有數百個或數千個 DO loop，但是其平行化的方法是一樣的。

```
      PROGRAM T2SEQ
      PARAMETER (NTOTAL=200)
      REAL*8       A(NTOTAL), B(NTOTAL), C(NTOTAL), D(NTOTAL), SUMA
C
C       test data generation and write to file    'input.dat'
C
      DO I=1,NTOTAL
        B(I)=3.D0/DFLOAT(I)+1.0
        C(I)=2.D0/DFLOAT(I)+1.0
        D(I)=1.D0/DFLOAT(I)+1.0
      ENDDO
      OPEN(7,FILE='input.dat',FORM='UNFORMATTED')
      WRITE(7) B
      WRITE(7) C
      WRITE(7) D
      CLOSE(7)
C
C       read    'input.dat' , compute and write out the result
C
      OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
      READ (7) B
```

```
      READ (7) C
      READ (7) D
      SUMA=0.0
      DO I=1,NTOTAL
        A(I)=B(I)+C(I)*D(I)
        SUMA=SUMA+A(I)
      ENDDO
      WRITE(*,101) (A(I),I=1,NTOTAL,5)
101   FORMAT(10F8.3)
      WRITE(*,102) SUMA
102   FORMAT('SUM of array A =', E15.5)
      STOP
      END
```

循序程式 T2SEQ 的測試結果如下 ：

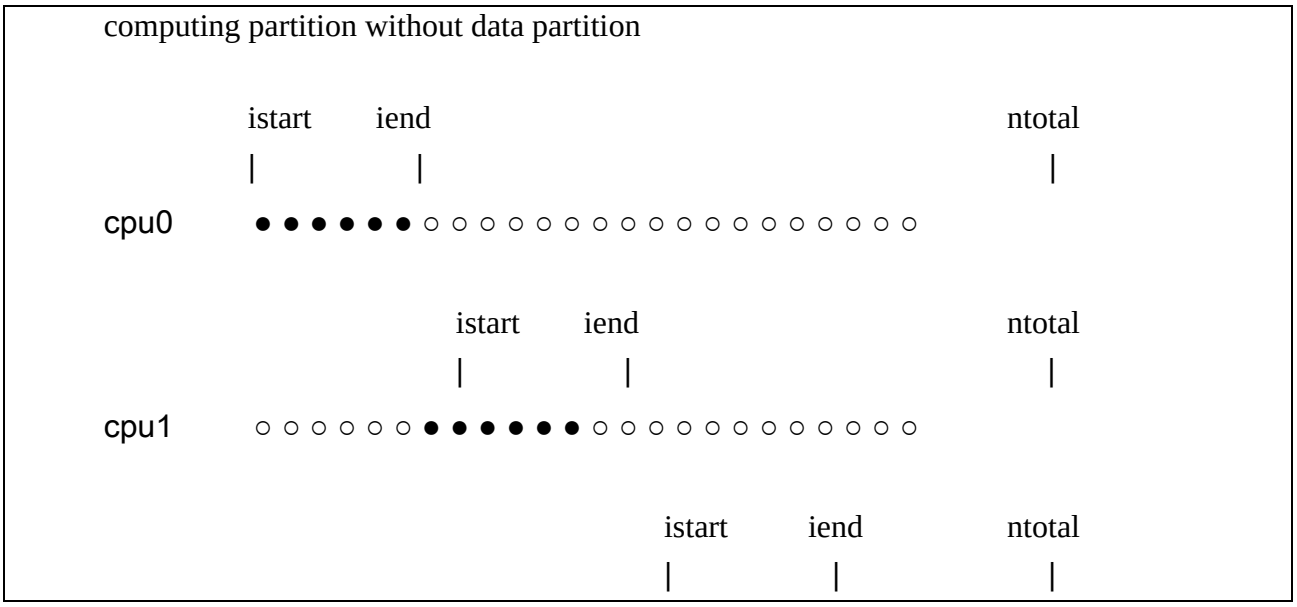
```
  10.000   3.056   2.562   2.383   2.290   2.234   2.196   2.168   2.148   2.131
   2.118   2.108   2.099   2.091   2.085   2.079   2.074   2.070   2.066   2.063
   2.060   2.057   2.054   2.052   2.050   2.048   2.046   2.044   2.043   2.041
   2.040   2.039   2.037   2.036   2.035   2.034   2.033   2.032   2.031   2.031
SUM of array A =        .43855E+03
```

2.3 資料不切割的平行程式 T2CP

平行程式的切割 (decomposition/partition) 方式有兩種。一種是計算切割而資料不切割，另外一種是計算和資料都切割。前一種切割方式不能夠節省記憶體的使用量是其缺點，但是陣列的描述與循序版 (sequential version) 完全相同，程式容易閱讀也容易維護是其優點。後一種切割方式能夠節省記憶體的使用量是其最大優點，但是陣列的描述與循序版差異較大，程式的閱讀與維護比較困難是其缺點。

如何將循序程式 T2SEQ 平行化呢? 這一節先介紹 '計算切割而資料不切割' 的方法，2.5 節再介紹 '計算及資料同時切割' 的方法。

假如 T2SEQ 程式要在四個 CPU 上平行計算而資料不切割時，就把一維陣列 A、B、C、D 均分為四段，各個 CPU 負責計算其中的一段，分工合作完成整個計算工作。此處是利用一個 STARTEND 副程式來計算各個 CPU 負責計算段落的起迄 index。它是把第一段分給 CPU0，第二段分給 CPU1，第三段分給 CPU2，餘類推。如圖 2.1 所示 :



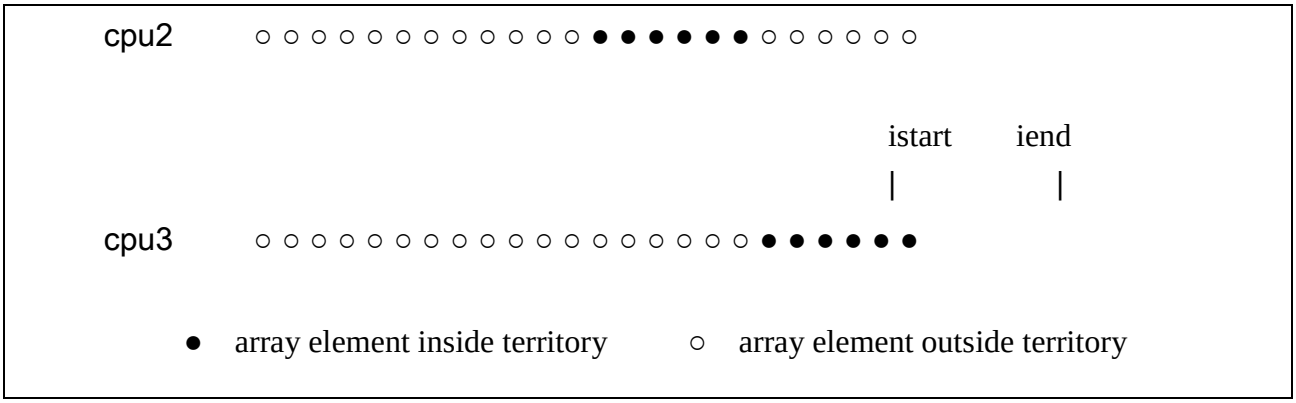


圖 2.1 計算切割而資料不切割的示意圖

圖 2.1 裏符號●代表轄區內的陣列元素，符號○代表轄區外的陣列元素，各個 CPU 負責計算的範圍是從該 CPU 的 istart 到 iend。

由於 MPI 1.2 版不具備平行輸出入（Parallel I/O）的功能，所以輸入資料由 CPU0（MYID 值為零）讀入後，利用一個 DO loop 分段傳送（MPI_SEND）給其他三個 CPU，而其他三個 CPU（MYID 值大於零）則接收由 CPU0 送來給該 CPU 的陣列片段。請留意，傳送不同的陣列片段要使用不同的資料標籤（ITAG），每一個 MPI_SEND 一定有一個對應的 MPI_RECV。每一個 CPU 算完自己負責的段落後，把計算的結果（陣列 A 的一部份）傳送給 CPU0，CPU0 利用一個 DO loop 把其他三個 CPU 送來的陣列片段逐一接收下來。然後由 CPU0 單獨計算整個 A 陣列各個元素的和 SUMA，再把 A 陣列和 SUMA 列印出來。

```
      PROGRAM T2CP
C
C     computing partition without data partition
C
      INCLUDE    'mpif.h'
      PARAMETER (NTOTAL=200)
```

```
      REAL*8         A(NTOTAL), B(NTOTAL), C(NTOTAL), D(NTOTAL), SUMA
      INTEGER      NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1                  COMM, GCOUNT(0:31), GSTART(0:31), GEND(0:31)

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL STARTEND ( NPROC, 1, NTOTAL, GSTART, GEND, GCOUNT)
      ISTART=GSTART(MYID)
      IEND=GEND(MYID)
      PRINT *,' NPROC,MYID,ISTART,IEND=',    NPROC,MYID,ISTART,IEND
      COMM=MPI_COMM_WORLD
C
C     Read input data and distribute data to each processor
C
      IF (MYID.EQ.0) THEN
        OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
        READ (7) B
        READ (7) C
        READ (7) D
        DO IDEST=1,NPROC-1
          ISTART1=GSTART(IDEST)
          KOUNT1=GCOUNT(IDEST)
          CALL MPI_SEND (B(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     &                              10, COMM, IERR)
          CALL MPI_SEND (C(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     &                               20, COMM, IERR)
          CALL MPI_SEND (D(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     &                               30, COMM, IERR)
        ENDDO
      ELSE
        KOUNT=GCOUNT(MYID)
        ISRC=0
        CALL MPI_RECV (B(ISTART), KOUNT, MPI_REAL8, ISRC, 10,
     1                        COMM, ISTATUS, IERR)
        CALL MPI_RECV (C(ISTART), KOUNT, MPI_REAL8, ISRC, 20,
     1                         COMM, ISTATUS, IERR)
        CALL MPI_RECV (D(ISTART), KOUNT, MPI_REAL8, ISRC, 30,
```

```
     1                          COMM, ISTATUS, IERR)
      ENDIF
C
C       Compute, collect computed result and write out the result
C
CCC   DO I=1,NTOTAL
      DO I=ISTART, IEND
        A(I)=B(I)+C(I)*D(I)
      ENDDO
      ITAG=110
      IF (MYID.NE.0) THEN
        KOUNT=GCOUNT(MYID)
        IDEST=0
        CALL MPI_SEND (A(ISTART), KOUNT, MPI_REAL8, IDEST, ITAG,
     &                          COMM, IERR)
      ELSE
        DO ISRC=1,NPROC-1
          ISTART1=GSTART(ISRC)
          KOUNT1=GCOUNT(ISRC)
          CALL MPI_RECV (A(ISTART1), KOUNT1, MPI_REAL8, ISRC, ITAG,
     &                           COMM, ISTATUS, IERR)
        ENDDO
      ENDIF
      IF(MYID.EQ.0) THEN
        WRITE(*,101) (A(I),I=1,NTOTAL,5)
        SUMA=0.0
        DO I=1,NTOTAL
          SUMA=SUMA+A(I)
        ENDDO
        WRITE(*,102) SUMA
      ENDIF
 101  FORMAT(10F8.3)
 102  FORMAT('SUM of array A =', E15.5)
      CALL MPI_FINALIZE (IERR)
      STOP
      END
      SUBROUTINE STARTEND(NPROC,IS1,IS2,GSTART,GEND,GCOUNT)
      INTEGER ID,NPROC,IS1,IS2,GSTART(0:31),GEND(0:31),GCOUNT(0:31)
```

```
      LENG=IS2-IS1+1
      IBLOCK=LENG/NPROC
      IR=LENG-IBLOCK*NPROC
      DO I=0,NPROC-1
        IF(I.LT.IR) THEN
          GSTART(I)=IS1+I*(IBLOCK+1)
          GEND(I)=GSTART(I)+IBLOCK
        ELSE
          GSTART(I)=IS1+I*IBLOCK+IR
          GEND(I)=GSTART(I)+IBLOCK-1
        ENDIF
        IF(LENG.LT.1) THEN
          GSTART(I)=1
          GEND(I)=0
        ENDIF
        GCOUNT(I)=GEND(I)-GSTART(I)+1
      ENDDO
      END
```

資料不切割平行程式 T2CP 的測試結果如下 ：

```
ATTENTION: 0031-408    4 nodes allocated by LoadLeveler, continuing...
   NPROC,MYID,ISTART,IEND= 4    2    101    150
   NPROC,MYID,ISTART,IEND= 4    1     51    100
   NPROC,MYID,ISTART,IEND= 4    3    151    200
   NPROC,MYID,ISTART,IEND= 4    0      1     50
   10.000     3.056     2.562     2.383     2.290     2.234     2.196     2.168     2.148     2.131
    2.118     2.108     2.099     2.091     2.085     2.079     2.074     2.070     2.066     2.063
    2.060     2.057     2.054     2.052     2.050     2.048     2.046     2.044     2.043     2.041
    2.040     2.039     2.037     2.036     2.035     2.034     2.033     2.032     2.031     2.031
SUM of array A =         .43855E+03
```

這一種平行程式的寫法叫做 SPMD (Single Program Multiple Data) 程式。這個程式在多顆 CPU 上平行時，每一顆 CPU 都是執行這一個程式，有些地方是用 rank (MYID) 來判斷要執行條件陳述(IF statement)裏的那一個區段，有些地方是用不同的 index 起迄位置來執行 DO loop 的

某一段落。沒有使用 index 或 rank 來區分執行的段落部份，則每一顆 CPU 都要執行。所以 T2CP 程式裏，每一顆 CPU 都會執行該程式一開始的下列陳述 ：

```
CALL MPI_INIT (IERR)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
CALL STARTEND (NPROC, 1, NTOTAL, GSTART,GEND,GCOUNT)
ISTART=GSTART(MYID)
IEND=GEND(MYID)
PRINT *,' NPROC,MYID,ISTART,IEND=', NPROC,MYID,ISTART,IEND
```

雖然執行的陳述完全一樣，但是每一個 CPU 得到的 MYID 值卻不一樣，所得到的 ISTART 和 IEND 也就隨之而異，從所列印出來的結果可以得到印證。這樣一個 SPMD 平行程式每一個 CPU 都有工作做。CPU0 會多做資料的讀和寫的工作。但是一般計算程式資料輸出入所佔的時間比例很小，所以各個 CPU 的負載相當均勻，也沒有 master CPU 和 slave CPU 的區別。

2.4 MPI_SCATTER，MPI_GATHER，MPI_REDUCE

MPI_SCATTER、MPI_GATHER、MPI_ALLGATHER、MPI_REDUCE、MPI_ALLREDUCE 都是屬於 '集體通訊' 類副程式。這一類的資料傳輸，凡是屬於同一個 communicator 的每一個 CPU 都要參與運作。所以使用這一種指令時，每一個 CPU 都必須叫用同一個副程式。

2.4.1 MPI_SCATTER，MPI_GATHER

MPI_SCATTER 的叫用格式如下 ：

```
      IROOT = 0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B, N, MPI_REAL8,
     &                                IROOT, MPI_COMM_WORLD, IERR)
```

MPI_SCATTER 是 IROOT CPU 把一個陣列 T 等分為 NPROC 段，(NPROC=參與平行計算的 CPU 數量)，每一段資料長度為 N，依 CPU id 的順序分送給每一個 CPU (包括 IROOT CPU 在內)。它是把第一段分給 CPU0，第二段分給 CPU1，第三段分給 CPU2，餘類推。如圖 2.2 所示：

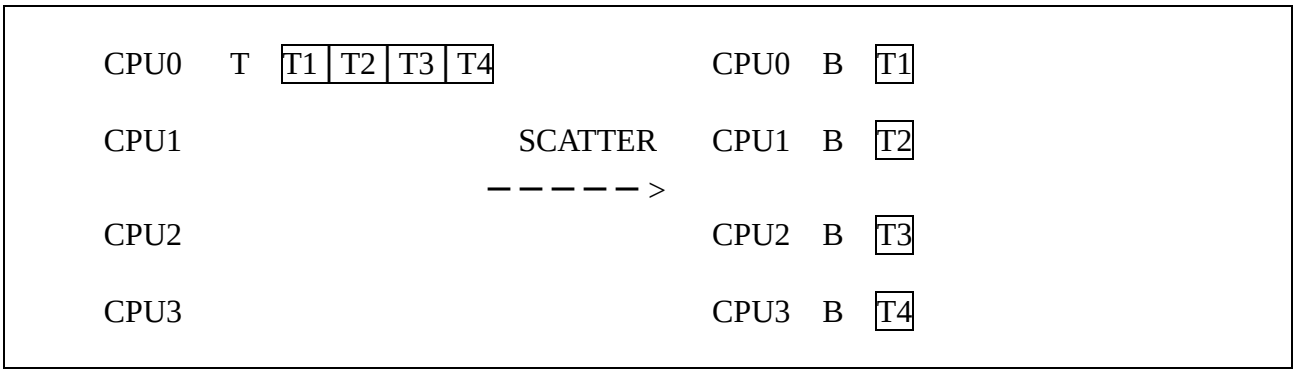


圖 2.2 MPI_SCATTER 示意圖

請注意每一段資料必須等長。其引數依序為

T　　是待送出陣列的起點

N　　是送給每一個 CPU 的資料數量

MPI_REAL8　　是待送出資料的類別

B　　是接收資料存放的起點，如果 N 值大於一時，B 必須是個陣列

N　　是接收資料的數量

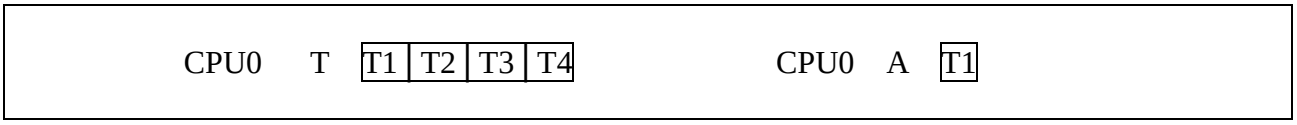
MPI_REAL8　　是接收資料的類別

IROOT　　是送出資料的 CPU id

MPI_GATHER 的叫用格式如下 ：

```
 IDEST = 0
 CALL MPI_GATHER (A, N, MPI_REAL8,   T, N, MPI_REAL8,
&                          IDEST, MPI_COMM_WORLD, IERR)
```

MPI_GATHER 與 MPI_SCATTER 的動作剛好相反，是 IDEST CPU 收集每一個 CPU 送給它的陣列 A，依 CPU id 的順序存入陣列 T 裏頭。也就是從 CPU0 收到的 N 個陣列元素存入 T 陣列的第一段，從 CPU1 收到的 N 個陣列元素存入 T 陣列的第二段，從 CPU2 收到的 N 個陣列元素存入 T 陣列的第三段，餘類推。如圖 2.3 所示 ：



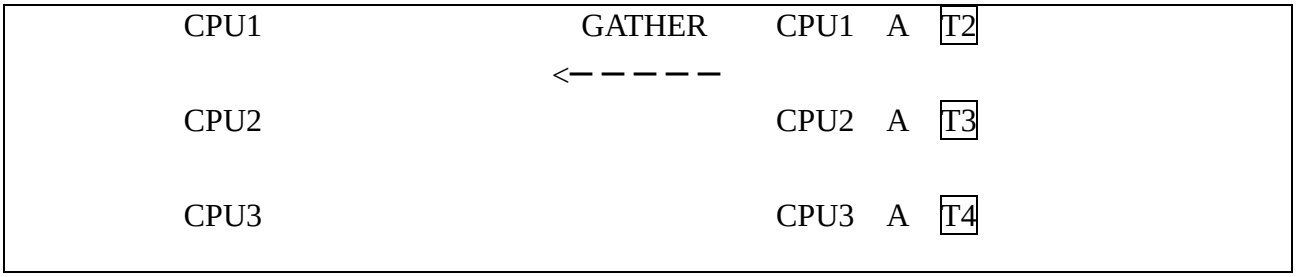


圖 2.3 MPI_GATHER 示意圖

請注意每一段資料必須等長。MPI_GATHER 的引數依序為

A 是待送出的資料起點，如果 N 值大於一時，A 必須是個陣列

N 是待送出資料的數量

MPI_REAL8 是待送出資料的類別

T 是接收資料存放的陣列起點

N 是接收來自各個 CPU 的資料數量

MPI_REAL8 是接收資料的類別

IDEST 是收集資料的 CPU id

MPI_ALLGATHER 的叫用格式如下 ：

```
CALL MPI_ALLGATHER (A, N, MPI_REAL8,   T, N, MPI_REAL8,
&                                  MPI_COMM_WORLD, IERR)
```

MPI_ALLGATHER 與 MPI_GATHER 的運作功能相似，MPI_GATHER 是把運作的結果存入指定的一個 CPU，而 MPI_ALLGATHER 則是把運作的結果存入每一個 CPU。

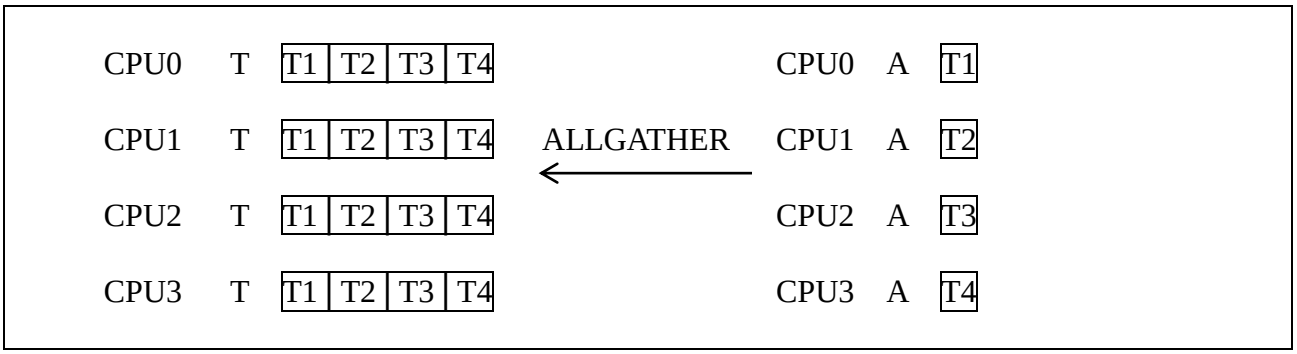


圖 2.4 MPI_ALLGATHER 示意圖

2.4.2 MPI_REDUCE, MPI_ALLREDUCE

另外一種集體資料傳輸功能叫作 '縮減運作' (reduction operation)，例如把各個 CPU 算出來的部份和 (partial sum) 加總，或找出各個 CPU 上某一個變數的最大值或最小值。

MPI_REDUCE 運作的結果只存放在指定的 CPU (IROOT)裏，MPI_ALLREDUCE 則是把運作的結果存放在每一個 CPU 裏。

MPI_REDUCE 和 MPI_ALLREDUCE 叫用格式如下 ：

```
 IROOT = 0
 CALL MPI_REDUCE ( SUMA, SUMALL, KOUNT, MPI_REAL8, MPI_SUM, IROOT,
&                  MPI_COMM_WORLD, IERR)

 CALL MPI_ALLREDUCE( SUMA, SUMALL, KOUNT, MPI_REAL8, MPI_SUM,
&                    MPI_COMM_WORLD, IERR)
```

引數 SUMA 是待運作 (累加) 的變數

SUMALL 是存放運作 (累加) 後的結果 (把各個 CPU 上的 SUMA 加總)

KOUNT 是待運作 (累加) 的資料個數

MPI_REAL8　　是 SUMA 和 SUMALL 的資料類別

MPI_SUM　　是運作涵數，可以選用的涵數如表 2.1

IROOT　　是存放運作結果的 CPU_id

MPI 指令	Operation	Data type
MPI_SUM MPI_PROD	sum 累加 product 乘積	MPI_INTEGER, MPI_REAL, MPI_REAL8, MPI_COMPLEX, MPI_COMPLEX16
MPI_MAX MPI_MIN	maximum 最大值 minimum 最小值	MPI_INTEGER, MPI_REAL, MPI_REAL8, MPI_DOUBLE_PRECISION
MPI_MAXLOC MPI_MINLOC	max value and location min value and location	MPI_2INTEGER, MPI_2REAL, MPI_2 DOUBLE_PRECISION
MPI_LAND MPI_LOR MPI_LXOR	logical AND logical OR logical exclusive OR	MPI_LOGICAL
MPI_BAND MPI_BOR MPI_BXOR	binary AND binary OR binary exclusive OR	MPI_INTEGER, MPI_BYTE

表 2.1 MPI Reduction Function

MPI_REDUCE 的運作方式如圖 2.5 所示，MPI_ALLREDUCE 的運作方式如圖 2.6 所示。

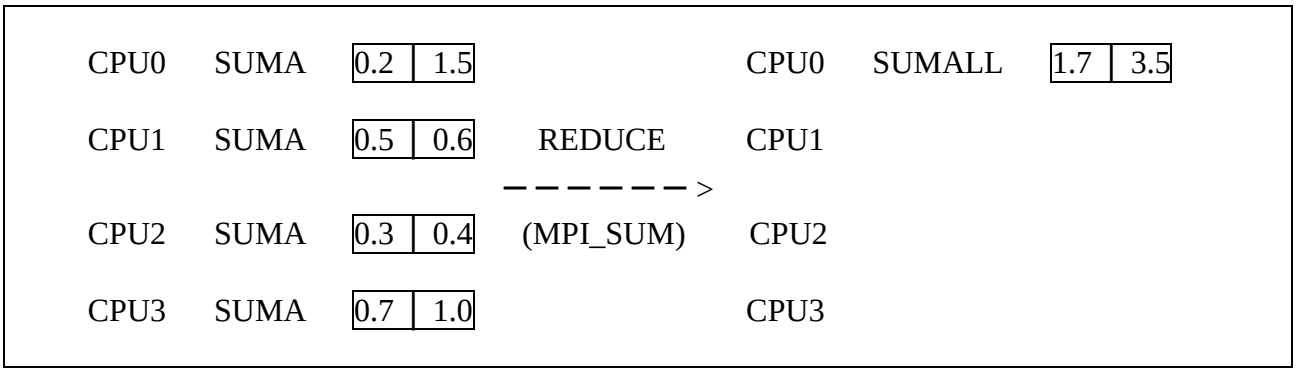


圖 2.5 MPI_REDUCE 示意圖

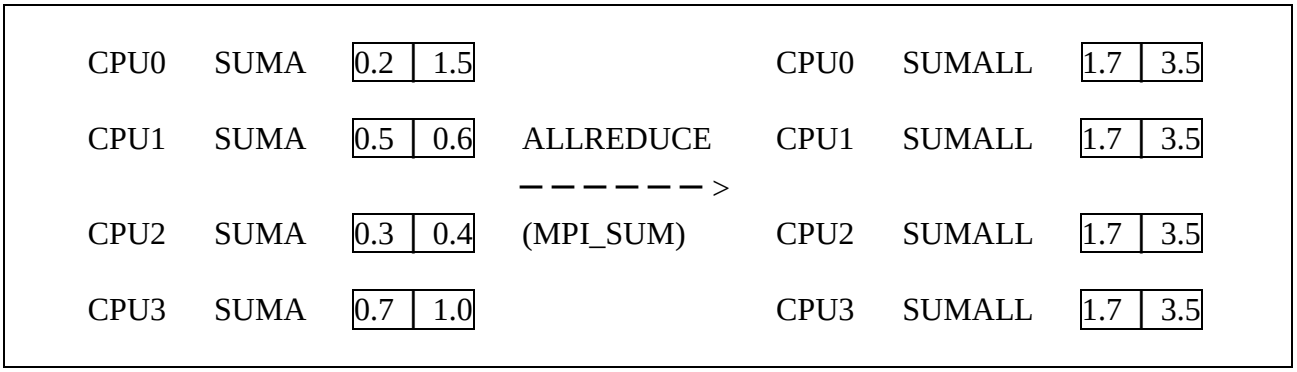


圖 2.6 MPI_ALLREDUCE 示意圖

此處 SUMALL 等於各個 CPU 上 SUMA 對應項目之和。

2.5 資料切割的平行程式 T2DCP

程式 T2DCP 是一個計算及資料都切割的平行程式。這個程式在 NP 個 CPU 上平行計算時，陣列 A、B、C、D 的長度只需要原來長度 NTOTAL 的 NP 分之一。但是輸入資料 B、C、D 陣列的長度是 NTOTAL，所以要再設定一個長度為 NTOTAL 的暫用陣列 T 來輪流存放讀入資料陣列 B、C、D 及存放輸出資料整個陣列 A。CPU0 每讀入一個輸入陣列就利用 MPI_SCATTER 分段分送給每一個 CPU。

```
      IROOT=0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B, N, MPI_REAL8,
     &                                IROOT, MPI_COMM_WORLD, IERR)
```

各個 CPU 把分配給它的陣列段落計算完畢之後，就利用 MPI_GATHER 把計算的結果 A 陣列片段送回給 CPU0。

```
      IDEST=0
      CALL MPI_GATHER (A, N, MPI_REAL8,    T, N, MPI_REAL8,
     &                                IDEST, MPI_COMM_WORLD, IERR)
```

比較 T2CP 和 T2DCP 兩個程式就可以看出利用 MPI_SCATTER，MPI_GATHER 的程式要簡潔得多。但是使用這種指令時分送給每一個 CPU 的陣列長度必須相等，而使用 MPI_SEND、MPI_RECV 指令時就沒有這種限制。

由於 Fortran 77 的 DIMENSION 必須是常數，所以必須在程式裏先設定 CPU 的個數 NP，切割後的陣列長度 N 等於 NTOTAL / NP，使用 PARAMETER 陳述設定如下 ：

```
      PARAMETER (NTOTAL=200, NP=4, N=NTOTAL/NP)
```

此處 NTOTAL 必須能被 NP 整除。於是，陣列的 dimension 由 NTOTAL 改為 N :

```
      REAL*8    A(N), B(N), C(N), D(N), T(NTOTAL)
```

現在，每一個 CPU 執行的 DO loop 範圍是從 1 到 N，所以算出來的 SUMA 是 A 陣列 NTOTAL 個元素裏 NP 分之一個陣列元素的和，是為部份和 (partial sum)。所以此處叫用 MPI_REDUCE 把各個 CPU 的部份和 SUMA 加總，存放在 CPU0 的 SUMALL 裏。

```
      IROOT=0
      CALL MPI_REDUCE (SUMA, SUMALL, 1, MPI_REAL8, MPI_SUM,
     &                 IROOT, MPI_COMM_WORLD, IERR)
```

整個計算及資料都切割的平行程式 T2DCP 如下 :

```
      PROGRAM T2DCP
C
C     Data & Computational Partition Using MPI_SCATTER, MPI_GATHER
C     NP=4 must be modified when run on other than 4 processors
C
C index        1 2 3 . . . n     1 2 3 . . . n     1 2 3 . . . n     1 2 3 . . . n
C              |o o o o o o|     |o o o o o o|     |o o o o o o|     |o o o o o o|
C
C                  P0                P1                P2                P3
C
      PARAMETER (NTOTAL=200, NP=4, N=NTOTAL/NP)
      INCLUDE   'mpif.h'
      REAL*8    A(N), B(N), C(N), D(N), T(NTOTAL), SUMA, SUMALL

      INTEGER   NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND
```

```
      IF (MOD(NTOTAL,NP).NE.0) THEN
         PRINT *,' NTOTAL IS NOT DIVISIBLE BY NP, PROGRAM WILL STOP'
         PRINT *,' NTOTAL,NP=', NTOTAL,NP
         STOP
      ENDIF
      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK ( MPI_COMM_WORLD, MYID, IERR)
      PRINT *,' NPROC,MYID=', NPROC, MYID
C
C     Read and distribute input data
C
      IF (MYID.EQ.0) THEN
         OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
         READ (7) T              ! read array B
      ENDIF
      IROOT=0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B, N, MPI_REAL8,
     &                  IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
         READ (7) T              ! read array C
      ENDIF
      CALL MPI_SCATTER (T, N, MPI_REAL8,    C, N, MPI_REAL8,
     &                  IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
         READ (7) T              ! read array D
      ENDIF
      CALL MPI_SCATTER (T, N, MPI_REAL8,    D, N, MPI_REAL8,
     &                  IROOT, MPI_COMM_WORLD, IERR)
C
C     Compute, GATHER computed data, and write out the result
C
      SUMA=0.0
CCC   DO I=1,NTOTAL
      DO I=1,N
         A(I)=B(I)+C(I)*D(I)
         SUMA=SUMA+A(I)
```

```
      ENDDO
      IDEST=0
      CALL MPI_GATHER (A, N, MPI_REAL8,    T, N, MPI_REAL8,
     &                          IDEST, MPI_COMM_WORLD, IERR)
      CALL MPI_REDUCE (SUMA, SUMALL, 1, MPI_REAL8, MPI_SUM,
     &                          IDEST, MPI_COMM_WORLD, IERR)
      IF (MYID.EQ.0) THEN
         WRITE(*,101) (T(I),I=1,NTOTAL,5)
         WRITE(*,102) SUMALL
      ENDIF
101   FORMAT(10F8.3)
102   FORMAT('SUM of array A =', E15.5)
      CALL MPI_FINALIZE (MPI_ERR)
      STOP
      END
```

計算及資料都切割的平行程式 T2DCP 測試結果如下 ：

```
ATTENTION: 0031-408    4 nodes allocated by LoadLeveler, continuing...
   NPROC,MYID= 4    0
   NPROC,MYID= 4    1
   NPROC,MYID= 4    2
   NPROC,MYID= 4    3
   10.000     3.056     2.562     2.383     2.290     2.234     2.196     2.168     2.148     2.131
    2.118     2.108     2.099     2.091     2.085     2.079     2.074     2.070     2.066     2.063
    2.060     2.057     2.054     2.052     2.050     2.048     2.046     2.044     2.043     2.041
    2.040     2.039     2.037     2.036     2.035     2.034     2.033     2.032     2.031     2.031
SUM of array A =          .43855E+03
```

第三章　需要邊界資料交換的平行程式

最常見的平行程式就是在計算的過程中需要邊界資料交換的平行程式。這一章將利用一個需要邊界資料交換的循序程式使用 MPI 指令加以平行化，並比較其計算結果以資驗證。

3.1 節　將介紹兩個 MPI 指令 MPI_SENDRECV、MPI_BCAST。MPI_SENDRECV 是用來交換邊界資料，而 MPI_BCAST 則是用來分送輸入資料。

3.2 節　介紹需要邊界資料交換的循序程式 T3SEQ。

3.3 節　說明使用 MPI_SENDRECV、MPI_SEND、MPI_RECV 指令平行化循序程式 T3SEQ 而成為平行程式 T3CP。然後說明使用 MPI_BCAST 取代 T3CP 所使用的 MPI_SEND 和 MPI_RECV 的方法。T3CP 是計算切割而資料不切割的平行程式。

3.4 節　說明交換一個陣列元素的資料切割平行程式 T3DCP_1。

3.5 節　說明交換兩個陣列元素的資料切割平行程式 T3DCP_2。

3.1 MPI_SENDRECV, MPI_BCAST

MPI_SENDRECV 是屬於 '點對點通訊' 類副程式，而 MPI_BCAST 是屬於 '集體通訊' 類副程式。

3.1.1 MPI_SENDRECV

甲 CPU 送出一些資料給乙 CPU，又要接受丙 CPU 送來另外一些資料時，可以叫用一個 MPI_SENDRECV 副程式來做這兩件工作。其作用等於一個 MPI_SEND 加一個 MPI_RECV。下面這個指令是把轄區裏最後一個陣列元素送給右鄰 CPU，同時自左鄰 CPU 接受轄區外的前一個陣列元素。

```
 ITAG=110
 CALL MPI_SENDRECV (B(IEND),       ICOUNT, DATA_TYPE, R_NBR, ITAG,
1                   B(ISTARTM1), ICOUNT, DATA_TYPE, L_NBR, ITAG,
2                   MPI_COMM_WORLD, ISTATUS, IERR)
```

引數 B(IEND) 是要送出去的資料起點

ICOUNT 是要送出去的資料數量

DATA_TYPE 是要送出去的資料類別

R_NBR 是要送出去的目的地 CPU id (右鄰)

ITAG 是要送出去的資料標籤

B(ISTARTM1) 是要接收的陣列起點

ICOUNT 是要接收的資料數量

DATA_TYPE　　是要接收的資料類別

L_NBR　　是要接收的來源地 CPU id (左鄰)

ITAG　　是要接收的資料標籤

ISTATUS　　是叫用這個副程式執行後的狀況

3.1.2 MPI_BCAST

MPI_BCAST 是屬於 '集體通訊' 類副程式，BCAST 是 Broadcast 的縮寫。當你要把同一項資料傳送給屬於同一個 communicator 其他各個 CPU 時就可以叫用這一個副程式。 既然是'集體通訊' 類副程式，參與平行計算的每一個 CPU 都要叫用這一個副程式，不允許只有少數幾個 CPU 叫用它，而其他 CPU 不叫用它。

MPI_BCAST 的叫用格式如下 :

```
 IROOT=0
 CALL MPI_BCAST ( B, ICOUNT, DATA_TYPE, IROOT,
&                 MPI_COMM_WORLD, IERR)
```

引數 B　　是要送出去的資料起點，簡單變數或陣列名稱

ICOUNT　　是要送出去的資料數量

DATA_TYPE　　是要送出去的資料類別

IROOT　　是要送出資料的 CPU id

MPI_BCAST 的運作方式如圖 3.1 所示 :

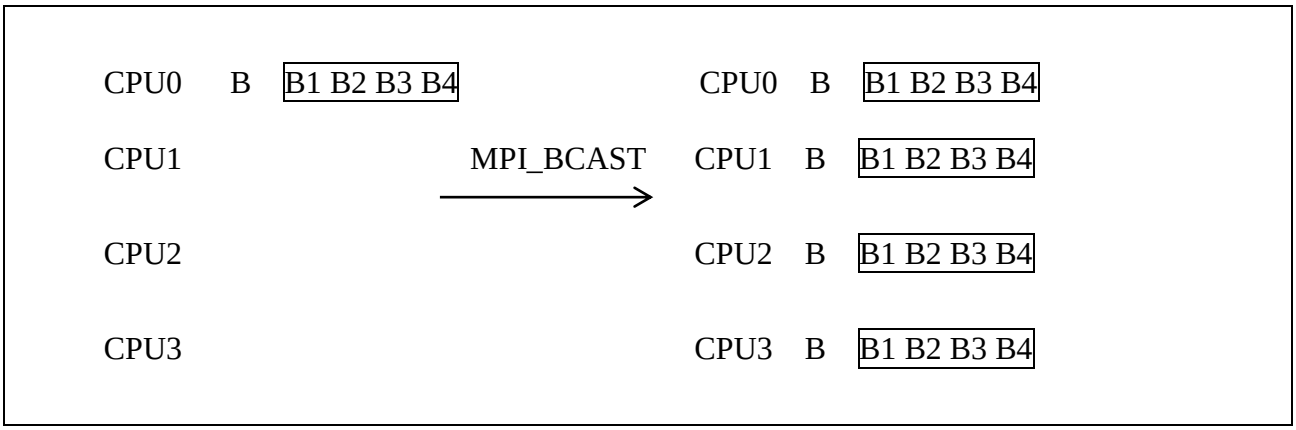


圖 3.1 MPI_BCAST 示意圖

3.2 邊界資料交換的循序程式 T3SEQ

T3SEQ 程式的 DO loop 裏，在算出等號左邊的 A(I) 時，要用到等號右邊的 C(I)、D(I)、和 B(I-1)、B(I)、B(I+1)。這在循序程式裏是奚鬆平常的事情，但是在計算分割的平行程式裏卻是一件大事。因為要用到該 CPU 轄區外 (outside territory) 的資料，這時就牽涉到邊界資料交換 (boundary data exchange) 的問題。另外還要找出 A 陣列各個元素的最大值 AMAX。

```
      PROGRAM T3SEQ
C
C       Boundary Data Exchange Program - Sequential Version
C
      PARAMETER (NTOTAL=200)
      REAL*8       A(NTOTAL), B(NTOTAL), C(NTOTAL), D(NTOTAL), AMAX

      OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
      READ (7) B
      READ (7) C
      READ (7) D
      AMAX=-1.D12
      DO I=2,NTOTAL-1
        A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
        AMAX=MAX(AMAX,A(I))
      ENDDO
      WRITE(*,101) (A(I),I=1,NTOTAL,5)
      WRITE(*,102) AMAX
101   FORMAT(10F8.3)
102   FORMAT('MAXIMUM VALUE OF A ARRAY is', E15.5)
      STOP
      END
```

循序程式 T3SEQ 的測試結果如下 :

.000 3.063 2.563 2.383 2.290 2.234 2.196 2.168 2.148 2.131
2.118 2.108 2.099 2.091 2.085 2.079 2.074 2.070 2.066 2.063
2.060 2.057 2.054 2.052 2.050 2.048 2.046 2.044 2.043 2.041
2.040 2.039 2.037 2.036 2.035 2.034 2.033 2.032 2.031 2.031
MAXIMUM VALUE OF ARRAY A is .57500E+01

3.3 資料不切割的邊界資料交換平行程式 T3CP

如何將循序程式 T3SEQ 平行化呢? 這一節先介紹計算切割而資料不切割的方法，3.4 節及 3.5 節再介紹計算及資料同時切割的方法。

程式 T3CP_1 也是利用副程式 STARTEND 來切割陣列，算出各個 CPU 轄區的 index 起迄點，它是把陣列的第一段分給 CPU0，第二段分給 CPU1，餘類推。如圖 3.2 所示 :

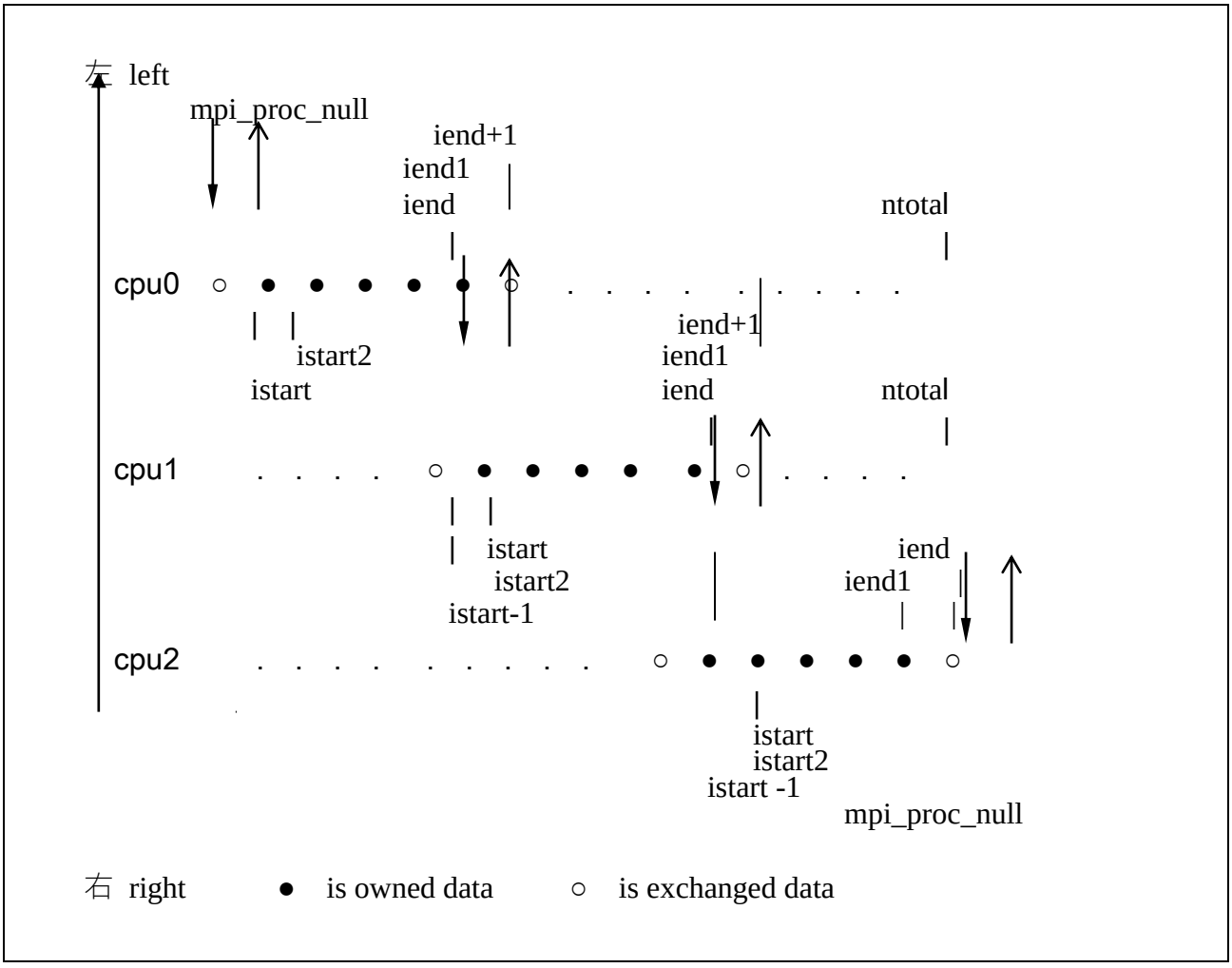


圖 3.2 資料不切割的邊界資料交換平行計算示意圖

圖 3.2 裏符號●代表轄區內的陣列元素，符號 ． 代表轄區外的陣列元素，符號○ 代表從左鄰右舍傳送過來的陣列元素，箭頭代表陣列元素傳送的方向。各個 CPU 負責計算的範圍是從該 CPU 的 istart 到 iend。程式 T3SEQ 的計算部份如下 ：

```
AMAX=-1.D12
DO I=2,NTOTAL-1
   A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
   AMAX=MAX(AMAX,A(I))
ENDDO
```

index I 是從 2 到 NTOTAL-1。在切割後，只有 CPU0 是從 2 開始，其他的 CPU 都是從 istart 開始，所以必須設一個變數 istart2 來解決這個問題 ：

```
ISTART2=ISTART
IF(MYID.EQ.0) ISTART2=2
```

而 loop 的終點只有最後一個 CPU 是到 NTOTAL-1，也就是 iend-1，其他的 CPU 都是到 iend，所以必須再設一個變數 iend1 來解決這個問題 ：

```
IEND1=IEND
IF(MYID.EQ.NPROC-1) IEND1=IEND-1
```

當 A(I)的 I 等於 istart 時，要用到 B(ISTART-1)。而當 A(I)的 I 等於 iend 時，要用到 B(IEND+1)，所以還必須設一個變數 istartm1 (istart minus 1) 和 iendp1 (iend plus 1) 來解決這 B 變數的 index 問題 ：

```
ISTARTM1=ISTART-1
IENDP1=IEND+1
```

在需要域外資料 (I-1 或 I+1 等) 的 DO loop 之前就要叫用 MPI_SENDRECV 來取得該項資料。不過在這之前先要知道該 CPU 的左鄰右舍是那一個 CPU，副程式 STARTEND 在切割陣列的 index 時是把第一段分給 CPU0，第二段分給 CPU1，第三段分給 CPU2，餘類推。所以一個 CPU 的左鄰 L_NBR 就是該 CPU 的 CPU id 減一，而其右鄰 R_NBR 就是該 CPU 的 CPU id 加一。只有第一個和最後一個 CPU 是例外，第一個 CPU 沒有左鄰，而最後一個 CPU 沒有右鄰，這時的左鄰右舍就要給他一個特定的名子叫做 MPI_PROC_NULL。這個 MPI_PROC_RULL 是在 mpif.h 檔裏已經設定的常數。

```
L_NBR=MYID-1
R_NBR=MYID+1
IF(MYID.EQ.0)              L_NBR=MPI_PROC_NULL
IF(MYID.EQ.NPROC-1)   R_NBR=MPI_PROC_NULL
```

現在來解決 B(I-1) 和 B(I+1) 的邊界資料交換問題，這需要兩個 MPI_SENDRECV，一個解決 B(I-1) 的資料交換，另外一個解決 B(I+1) 的資料交換。

先來解決 B(I-1) 的邊界資料交換問題。從圖 3.2 中的 CPU1 來看，它要送 B(IEND) 給右鄰 "當作右鄰的 B(ISTARTM1)" ，又要自左鄰取得 "左鄰的 B(IEND)" 做為它自己的 B(ISTARTM1)。如果要傳送的對象是 MPI_PROC_NULL 時，是沒有傳送動作發生的。每一個 CPU 都做同樣的動作，就解決了 B(ISTARTM1) 的邊界資料交換問題。也就是 ：

```
      ITAG=110
      CALL MPI_SENDRECV (B(IEND),          1, MPI_REAL8, R_NBR, ITAG,
     1                   B(ISTARTM1), 1, MPI_REAL8, L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
```

再來解決 B(I+1) 的邊界資料交換問題。從圖 3.2 中的 CPU1 來看，它要送 B(ISTART) 給左鄰 "當作左鄰的 B(IENDP1)" ，又要自右鄰取得 "右鄰的 B(ISTART)" 做為它自己的 B(IENDP1)。如果要傳送的對象是 MPI_PROC_NULL 時，是沒有傳送動作發生的。每一個 CPU 都做同樣的動作，就解決了 B(IENDP1) 的邊界資料交換問題。也就是 ：

```
      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 1, MPI_REAL8,    L_NBR, ITAG,
     1                   B(IENDP1), 1, MPI_REAL8,    R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
```

現在，每一個 CPU 執行的 DO loop 範圍是從 1 到 N，所以算出來的 AMAX 是 A 陣列 NTOTAL 個元素裏 NP 分之一個陣列元素的最大值，只是部份資料的最大值。所以此處叫用 MPI_ALLREDUCE 找出各個 CPU 轄區裏的最大值 AMAX 之中的最大值 GMAX (global maximum)，存放在每一個 CPU 裏。

```
      CALL MPI_ALLREDUCE ( AMAX, GMAX, 1, MPI_REAL8, MPI_MAX,
     1                     MPI_COMM_WORLD, IERR )
```

至於甚麼時候使用 reduce，甚麼時候使用 allreduce，視需要而定。每一個 CPU 都要用到 reduce 的結果時就要用 MPI_ALLREDUCE，只有一個 CPU 需要用到 reduce 的結果時只要用

MPI_REDUCE 就可以了，因為 MPI_ALLREDUCE 比較耗時間。

所以完整的邊界資料交換平行程式如下 ：

```
      PROGRAM T3CP
C
C     Boundary data exchange without data partition
C     Using MPI_SEND, MPI_RECV to distribute input data
C
      PARAMETER (NTOTAL=200)
      INCLUDE    'mpif.h'
      REAL*8     A(NTOTAL), B(NTOTAL), C(NTOTAL), D(NTOTAL), AMAX, GMAX

      INTEGER    NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1           L_NBR, R_NBR,
     2           GSTART(0:31), GEND(0:31), GCOUNT(0:31)

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL STARTEND (NPROC, 1, NTOTAL, GSTART, GEND,GCOUNT)
      ISTART=GSTART(MYID)
      IEND=GEND(MYID)
      PRINT *,' NPROC,MYID,ISTART,IEND=', NPROC,MYID,ISTART,IEND

      ISTARTM1=ISTART-1
      IENDP1=IEND+1
      ISTART2=ISTART
```

```
      IF(MYID.EQ.0) ISTART2=2
      IEND1=IEND
      IF(MYID.EQ.NPROC-1) IEND1=IEND-1
C
      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)            L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.NPROC-1) R_NBR=MPI_PROC_NULL
C
      IF(MYID.EQ.0) THEN
        OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
        READ (7) B
        READ (7) C
        READ (7) D
        DO IDEST=1,NPROC-1
          ISTART1=GSTART(IDEST)
          KOUNT1=GCOUNT(IDEST)
          ITAG=10
          CALL MPI_SEND ( B(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     1                    ITAG, MPI_COMM_WORLD, IERR)
          ITAG=20
          CALL MPI_SEND (C(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     1                    ITAG, MPI_COMM_WORLD, IERR)
          ITAG=30
          CALL MPI_SEND (D(ISTART1), KOUNT1, MPI_REAL8, IDEST,
     1                    ITAG, MPI_COMM_WORLD, IERR)
        ENDDO
       ELSE
        KOUNT=(IEND-ISTART+1)
        ISRC=0
        ITAG=10
        CALL MPI_RECV (B(ISTART), KOUNT, MPI_REAL8, ISRC, ITAG,
     1                  MPI_COMM_WORLD, ISTATUS, IERR)
        ITAG=20
        CALL MPI_RECV (C(ISTART), KOUNT, MPI_REAL8, SRC, ITAG,
     1                  MPI_COMM_WORLD, ISTATUS, IERR)
        ITAG=30
        CALL MPI_RECV (D(ISTART), KOUNT, MPI_REAL8, ISRC, ITAG,
```

```
     1                          MPI_COMM_WORLD, ISTATUS, IERR)
      ENDIF
C
C       Exchange data outside the territory
C
      ITAG=110
      CALL MPI_SENDRECV (B(IEND),         1, MPI_REAL8, R_NBR, ITAG,
     1                   B(ISTARTM1), 1, MPI_REAL8, L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 1, MPI_REAL8, L_NBR, ITAG,
     1                   B(IENDP1), 1, MPI_REAL8, R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
C
C       Computate, gather and write out the computed result
C
      AMAX=-1.0D12
CCC   DO I=2,NTOTAL-1
      DO I=ISTART2,IEND1
        A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
        AMAX=MAX(AMAX,A(I))
      ENDDO
      ITAG=110
      IF(MYID.NE.0) THEN
        KOUNT=(IEND-ISTART+1)
        IDEST=0
        CALL MPI_SEND (A(ISTART), KOUNT, MPI_REAL8, IDEST, ITAG,
     1                 MPI_COMM_WORLD, IERR)
      ELSE
       DO ISRC=1,NPROC-1
          ISTART1=GSTART(ISRC)
          KOUNT1=GCOUNT(ISRC)
          CALL MPI_RECV (A(ISTART1), KOUNT1, MPI_REAL8, ISRC, ITAG,
     1                   MPI_COMM_WORLD, ISTATUS, IERR)
        ENDDO
      ENDIF
      CALL MPI_ALLREDUCE (AMAX, GMAX, 1, MPI_REAL8, MPI_MAX,
     1                    MPI_COMM_WORLD, IERR)
```

```
      IF(MYID.EQ.0) THEN
        WRITE(*,101) (A(I),I=1,NTOTAL,5)
        WRITE(*,102) GMAX
      ENDIF
101   FORMAT(10F8.3)
102   FORMAT('MAXIMUM VALUE of ARRAY A is', E15.5)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

計算切割而資料不切割的邊界資料交換平行程式 T3CP_1 的測試結果如下 ：

```
ATTENTION: 0031-408   4 nodes allocated by LoadLeveler, continuing...
  NPROC,MYID,ISTART,IEND= 4   3   151   200
  NPROC,MYID,ISTART,IEND= 4   2   101   150
  NPROC,MYID,ISTART,IEND= 4   1    51   100
  NPROC,MYID,ISTART,IEND= 4   0     1    50
    .000    3.063    2.563    2.383    2.290    2.234    2.196    2.168    2.148    2.131
   2.118    2.108    2.099    2.091    2.085    2.079    2.074    2.070    2.066    2.063
   2.060    2.057    2.054    2.052    2.050    2.048    2.046    2.044    2.043    2.041
   2.040    2.039    2.037    2.036    2.035    2.034    2.033    2.032    2.031    2.031
MAXIMUM VALUE OF ARRAY A is          .57500E+01
```

在陣列不切割的平行程式裏，CPU0 讀入資料之後也可以使用 MPI_BCAST 把整個陣列傳送給各個 CPU。這樣做程式寫起來比較簡單，但是卻傳送了許多額外的資料。利敝得失端看兩種傳送方式所花費總時數的多寡而定，通常一次傳送大量資料比分多次傳送小量資料來得省時。如何取得平行程式執行時的時間資訊將在第四章裏介紹。下列程式片段就是使用 MPI_BCAST 取代 T3CP 所使用的 MPI_SEND 和 MPI_RECV 來分送輸入資料的結果。

```
C
C       Read and distribute input data
C
        IF(MYID.EQ.0) THEN
           OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
           READ (7) B
           READ (7) C
           READ (7) D
        ENDIF
        IROOT=0
        CALL MPI_BCAST ( B, NTOTAL, MPI_REAL8, IROOT,
       1                 MPI_COMM_WORLD, IERR)
        CALL MPI_BCAST( C, NTOTAL, MPI_REAL8, IROOT,
       1                 MPI_COMM_WORLD, IERR)
        CALL MPI_BCAST( D, NTOTAL, MPI_REAL8, IROOT,
       1                 MPI_COMM_WORLD, IERR)
```

3.4 資料切割的邊界資料交換平行程式(一) T3DCP_1

計算及資料同時切割時，如果是在 NP 個 CPU 上平行計算，則陣列的長度 N 只需要原來長度 NTOTAL 的 NP 分之一。此時，NTOTAL 必須能被 NP 整除。再加上前後各保留一個界外陣列元素的存放位置，則所需的陣列長度為 N+2，本文主張其 DIMENSION 採用 (0:N+1) 的方式來撰寫，則其轄區內資料是存放在 index 1 到 N。於是陣列長度的設定為 ：

```
REAL*8      A(0:N+1), B(0:N+1), C(0:N+1), D(0:N+1), T(NTOTAL), AMAX, GMAX
```

如圖 3.3 所示 ：

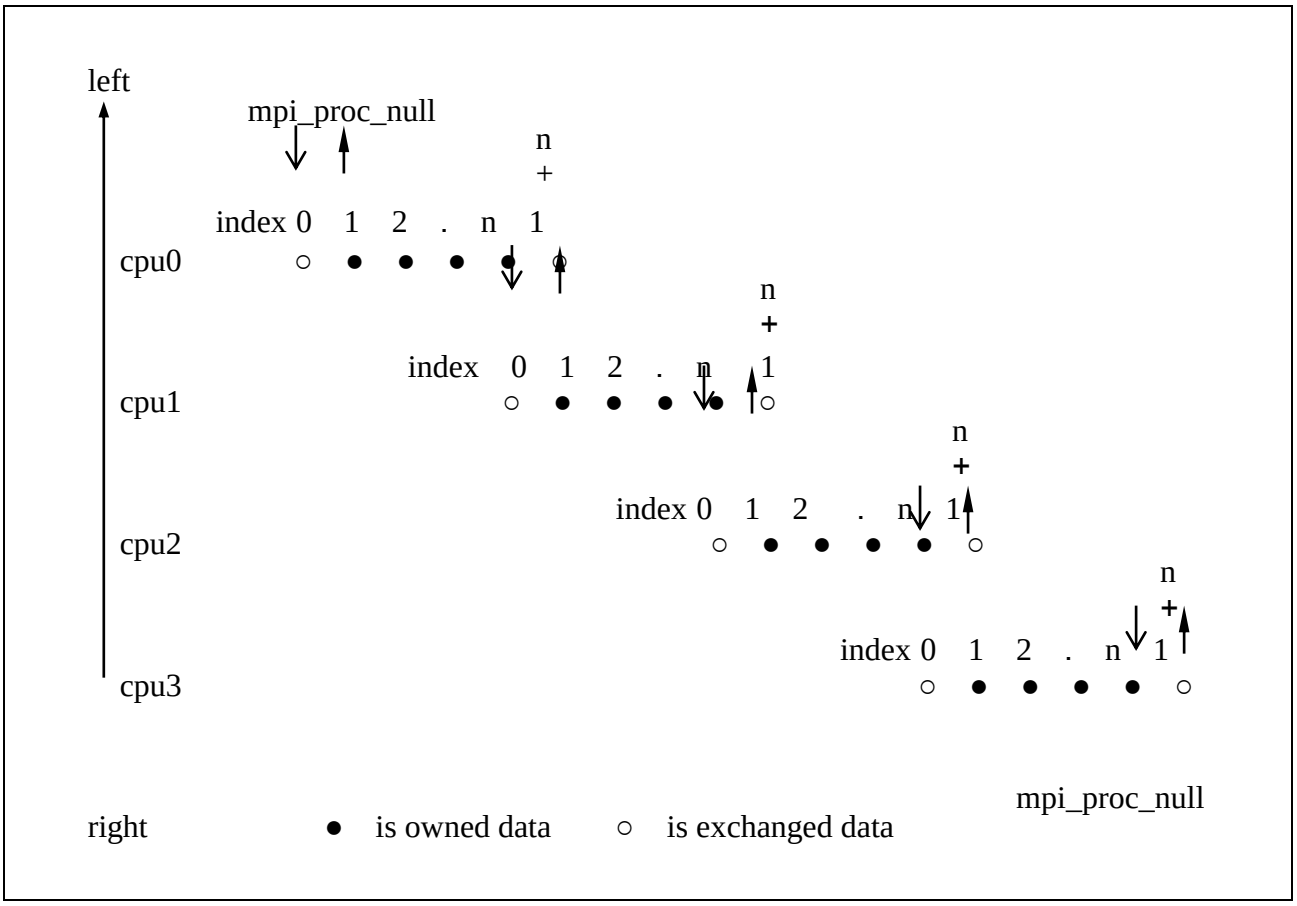


圖 3.3 資料切割且交換一個邊界資料的平行計算示意圖

這樣一來，每一個 CPU 的 DO loop 起點為 1，終點為 N，於是 ：

```
ISTART=1
IEND=N
```

第一個 CPU 的 DO loop 起點為 2，其他 CPU 的 DO loop 起點為 1，最後一個 CPU 的 DO loop 終點為 N-1，其他 CPU 的 DO loop 終點為 N，也就是 ：

```
ISTART2=ISTART
IF(MYID.EQ.0) ISTART2=2
IEND1=IEND
IF(MYID.EQ.NPROC-1) IEND1=IEND-1
```

每一個 CPU 要把轄區內最後一個陣列元素送給右邊的 CPU，這個陣列元素的位置是 iend，同時接受來自左邊 CPU 的陣列元素是放在 istart-1。所以 ：

```
ISTARTM1=ISTART-1

ITAG=110
CALL MPI_SENDRECV (B(IEND),         1, MPI_REAL8, R_NBR, ITAG,
1                               B(ISTARTM1), 1, MPI_REAL8, L_NBR, ITAG,
2                               MPI_COMM_WORLD, ISTATUS, IERR)
```

每一個 CPU 也要把轄區內第一個陣列元素送給左邊的 CPU，這個陣列元素的位置是 istart，同時接受來自右邊 CPU 的一個陣列元素是放在 iend+1。所以 ：

```
      IENDP1=IENDP1+1

      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 1, MPI_REAL8, L_NBR, ITAG,
     1                   B(IENDP1), 1, MPI_REAL8, R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
```

NTOTAL 個元素的 B、C、D 陣列資料讀入陣列 T 之後，在用 MPI_SCATTER 分送給各個 CPU 時，B、C、D 陣列的 DIMENSION 是從零開始，而輸入資料是從 1 開始存放。所以在 MPI_SCATTER 裏必需敘明要從 B(1)、C(1)、D(1)開始存放。MPI_GATHER 時亦然。

```
      IROOT=0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B(1), N, MPI_REAL8,
     1                  IROOT, MPI_COMM_WORLD, MPI_ERR)

      CALL MPI_GATHER (A(1), N, MPI_REAL8,    T, N, MPI_REAL8,
     1                 IROOT, MPI_COMM_WORLD, MPI_ERR)
```

計算及資料都切割的邊界資料交換平行程式 T3DCP_1 如下 ：

```
      PROGRAM T3DCP_1
C
C       Data Partition Using MPI_SCATTER, MPI_GATHER    with -1,+1 data exchange
C
      PARAMETER (NTOTAL=200, NP=4, N=NTOTAL/NP)
      INCLUDE    'mpif.h'
      REAL*8       A(0:N+1), B(0:N+1), C(0:N+1), D(0:N+1), T(NTOTAL),
     &             AMAX, GMAX
      INTEGER    NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1           L_NBR, R_NBR

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
```

```
      IF (NP.NE.NPROC) THEN
         PRINT *,' NP NOT EQUAL TO NPROC, PROGRAM WILL STOP'
         PRINT *,' NP,NPROC=', NP,NPROC
         STOP
      ENDIF

      ISTART=1
      IEND=N
      ISTARTM1=ISTART-1
      IENDP1=IEND+1
      ISTART2=1
      IF(MYID.EQ.0) ISTART2=2
      IEND1=N
      IF(MYID.EQ.NPROC-1) IEND1=N-1

      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)          L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.NPROC-1) R_NBR=MPI_PROC_NULL

C
C     Read & distribute input data
C
      IF(MYID.EQ.0) THEN
         OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
         READ (7) T                    ! read array B
      ENDIF
      IROOT=0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B(1), N, MPI_REAL8,
     1                  IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
         READ (7) T                    ! read array C
      ENDIF
      CALL MPI_SCATTER (T, N, MPI_REAL8,    C(1), N, MPI_REAL8,
     1                  IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
         READ (7) T                    ! read array D
      ENDIF
```

```
      CALL MPI_SCATTER (T, N, MPI_REAL8,    D(1), N, MPI_REAL8,
     1                                IROOT, MPI_COMM_WORLD, IERR)
C
C       Boundary data exchange
C
      ITAG=110
      CALL MPI_SENDRECV (B(IEND),         1, MPI_REAL8, R_NBR, ITAG,
     1                   B(ISTARTM1), 1, MPI_REAL8, L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 1, MPI_REAL8, L_NBR, ITAG,
     1                   B(IENDP1), 1, MPI_REAL8, R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
C
C       Compute and output result
C
      AMAX=-1.0D12
CCC   DO I=2,NTOTAL-1
      DO I=ISTART2,IEND1
        A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
        AMAX=MAX(AMAX,A(I))
      ENDDO

      CALL MPI_GATHER (A(1), N, MPI_REAL8,    T, N, MPI_REAL8,
     1                          IROOT, MPI_COMM_WORLD, IERR)
      CALL MPI_ALLREDUCE (AMAX, GMAX, 1, MPI_REAL8, MPI_MAX,
     1                    MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
        WRITE(*,101) (T(I),I=1,NTOTAL,5)
        WRITE(*,102)    GMAX
      ENDIF
101   FORMAT(10F8.3)
102   FORMAT('MAXIMUM VALUE OF ARRAY A is', E15.5)
      CALL MPI_FINALIZE (MPI_ERR)
      STOP
      END
```

計算及資料都切割的邊界資料交換平行程式 T3DCP_1 的測試結果如下 ：

```
ATTENTION: 0031-408  4 nodes allocated by LoadLeveler, continuing...
    .000    3.063    2.563    2.383    2.290    2.234    2.196    2.168    2.148    2.131
  2.118    2.108    2.099    2.091    2.085    2.079    2.074    2.070    2.066    2.063
  2.060    2.057    2.054    2.052    2.050    2.048    2.046    2.044    2.043    2.041
  2.040    2.039    2.037    2.036    2.035    2.034    2.033    2.032    2.031    2.031
MAXIMUM VALUE OF ARRAY A is         .57500E+01
```

3.5 資料切割的邊界資料交換平行程式 (二) T3DCP_2

如果 3.4 節裏程式 T3DCP_1 的 DO loop 需要用到兩個鄰近陣列元素時，如下列所示 ：

```
 DO I=3,NTOTAL-2
    A(I)=C(I)*D(I)+(    B(I-2)+2.0*B(I-1)+2.0*B(I)
&                       +2.0*B(I+1)+B(I+2)      )*0.125
 ENDDO
```

其平行化方法與程式 T3DCP_1 相同。只要把切割後陣列的 DIMENSION 前後各加兩個陣列元素，把 DIMENSION 改為 (-1:N+2)，其轄區內資料仍然是存放在 index 1 到 N。

```
 REAL*8      A(-1:N+2), B(-1:N+2), C(-1:N+2), D(-1:N+2), T(NTOTAL),
&            AMAX, GMAX

 ISTART=1
 IEND=N
```

如圖 3.4 所示 ：

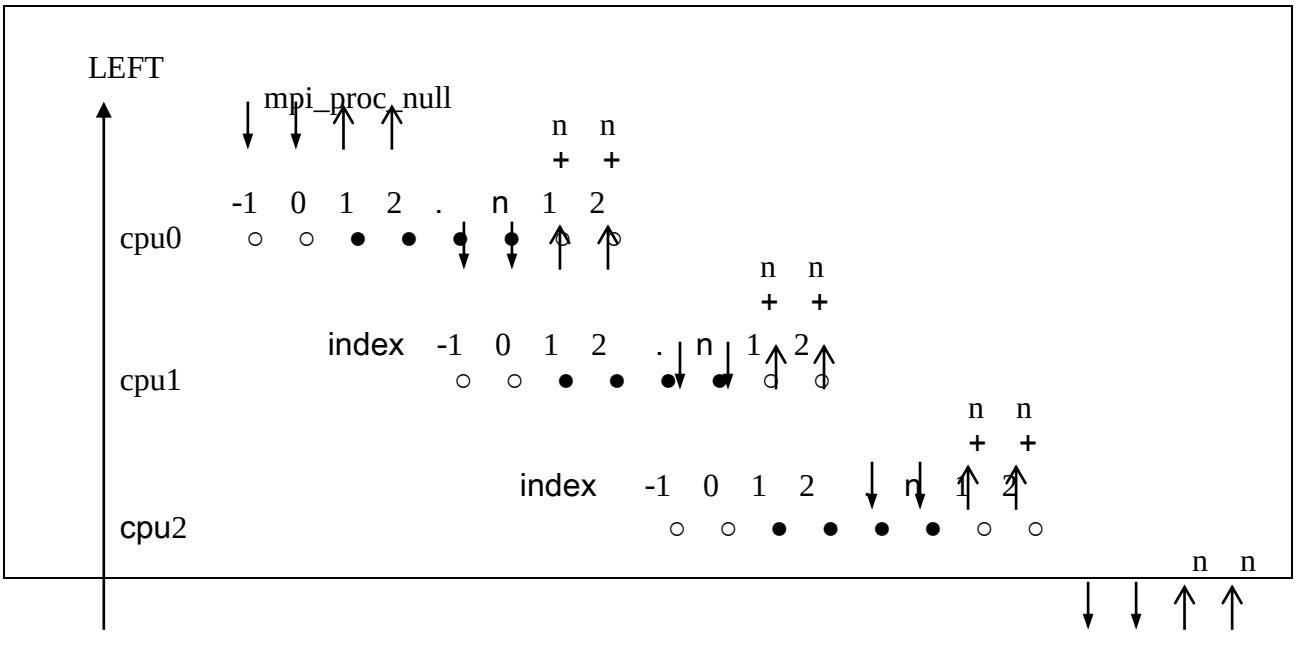


圖 3.4 資料切割且交換兩個邊界資料的平行計算示意圖

DO loop 的起迄 index 也要加以修改。第一個 CPU 的起點是 3，最後一個 CPU 的終點是 NTOTAL-2，所以：

```
ISTART3=1
IF(MYID.EQ.0) ISTART3=3
IEND2=IEND
IF(MYID.EQ.NPROC-1) IEND2=IEND-2
```

當然，邊界資料的交換量也要由一個改為兩個。每一個 CPU 要把轄區內最後兩個陣列元素送給右邊的 CPU，這兩個陣列元素的起點是 iend-1，同時要接受來自左邊 CPU 的兩個陣列元素，從 istart-2 放起。所以：

```
IENDM1=IEND-1
ISTARTM2=ISTART-2

ITAG=110
CALL MPI_SENDRECV (B(IENDM1),     2, MPI_REAL8, R_NBR, ITAG,
1                  B(ISTARTM2), 2, MPI_REAL8, L_NBR, ITAG,
2                  MPI_COMM_WORLD, ISTATUS, IERR)
```

每一個 CPU 也要把轄區內最前面兩個陣列元素送給左邊的 CPU，這兩個陣列元素的起點是

istart，同時要接受來自右邊 CPU 的兩個陣列元素，從 iend+1 放起。所以 ：

```
      IENDP1=IEND+1

      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 2, MPI_REAL8, L_NBR, ITAG,
     1                   B(IENDP1), 2, MPI_REAL8, R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
```

交換邊界兩個陣列元素的平行程式如下 ：

```
      PROGRAM T3DCP_2
C
C     Balanced Data Partition Using MPI_SCATTER, MPI_GATHER
C     with -1,-2,+1,+2 data exchange
C
      PARAMETER (NTOTAL=200,NP=4,N=NTOTAL/NP)
      INCLUDE   'mpif.h'
      REAL*8    A(-1:N+2), B(-1:N+2), C(-1:N+2), D(-1:N+2), T(NTOTAL),
     &          AMAX, GMAX
      INTEGER   NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1          L_NBR, R_NBR

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
```

```
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      IF (NP.NE.NPROC) THEN
         PRINT *,' NP NOT EQUAL TO NPROC, PROGRAM WILL STOP'
         PRINT *,' NP,NPROC=', NP,NPROC
         STOP
      ENDIF

      ISTART=1
      ISTARTM2=ISTART-2
      ISTART3=1
      IF(MYID.EQ.0) ISTART3=3

      IEND=N
      IEND2=IEND
      IF(MYID.EQ.NPROC-1) IEND2=IEND-2
      IENDM1=IEND-1
      IENDP1=IEND+1
      IEND1=N
      IF(MYID.EQ.NPROC-1) IEND1=N-1

      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)             L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.NPROC-1) R_NBR=MPI_PROC_NULL
C
C     Read & distribute input data
C
      IF(MYID.EQ.0) THEN
         OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
         READ (7) T                    ! read array B
      ENDIF
      IROOT=0
      CALL MPI_SCATTER (T, N, MPI_REAL8,    B(1), N, MPI_REAL8,
     1                           IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
         READ (7) T                    ! read array C
      ENDIF
      CALL MPI_SCATTER (T, N, MPI_REAL8,    C(1), N, MPI_REAL8,
```

```
     1                          IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
        READ (7) T                    ! read array D
      ENDIF
      CALL MPI_SCATTER (T, N, MPI_REAL8,    D(1), N, MPI_REAL8,
     1                          IROOT, MPI_COMM_WORLD, IERR)
C
C       Boundary data exchange
C
      ITAG=110
      CALL MPI_SENDRECV (B(IENDM1),    2, MPI_REAL8, R_NBR, ITAG,
     1                           B(ISTARTM2), 2, MPI_REAL8, L_NBR, ITAG,
     2                           MPI_COMM_WORLD, ISTATUS, IERR)
      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 2, MPI_REAL8, L_NBR, ITAG,
     1                           B(IENDP1), 2, MPI_REAL8, R_NBR, ITAG,
     2                           MPI_COMM_WORLD, ISTATUS, IERR)
C
C       Compute and output result
C
      AMAX=-1.0D12
CCC   DO I=3,NTOTAL-2
      DO I=ISTART3,IEND2
        A(I)=C(I)*D(I)+( B(I-2)+2.0*B(I-1)+2.0*B(I)
     &                          +2.0*B(I+1)+B(I+2) )*0.125
        AMAX=MAX(AMAX,A(I))
      ENDDO
      IDEST=0
      CALL MPI_GATHER (A(1), N, MPI_REAL8,    T, N, MPI_REAL8,
     1                          IDEST, MPI_COMM_WORLD, IERR)
      CALL MPI_ALLREDUCE (AMAX, GMAX, 1, MPI_REAL8, MPI_MAX,
     1                           MPI_COMM_WORLD, IERR)
      AMAX=GMAX
      IF(MYID.EQ.0) THEN
        WRITE(*,101) (T(I),I=1,NTOTAL,5)
        WRITE(*,102) AMAX
      ENDIF
  101   FORMAT(10F8.3)
```

```
  102   FORMAT('MAXIMUM VALUE OF ARRAY A is ', E15.5)
        CALL MPI_FINALIZE (IERR)
        STOP
        END
```

交換邊界兩個陣列元素的平行程式 T3DCP_2 的測試結果如下 ：

```
ATTENTION: 0031-408   4 nodes allocated by LoadLeveler, continuing...
     .000    3.078    2.565    2.384    2.291    2.234    2.196    2.168    2.148    2.131
    2.118    2.108    2.099    2.091    2.085    2.079    2.074    2.070    2.066    2.063
    2.060    2.057    2.054    2.052    2.050    2.048    2.046    2.044    2.043    2.041
    2.040    2.039    2.037    2.036    2.035    2.034    2.033    2.032    2.031    2.031
MAXIMUM VALUE OF ARRAY A is         .44847E+01
```

第四章　格點數不能整除的平行程式

本章將探討格點數 (grid points) 不能被參與平行計算的 CPU 數目整除時的處理方法。所謂格點數就是陣列的 dimension。

4.1 節 循序程式 T4SEQ 裏陣列的 dimension 是 161，只能夠被 7 和 23 整除。

4.2 節 介紹 MPI_SCATTERV 和 MPI_GATHERV 兩個 '集體通訊' 類副程式，其功能與 MPI_SCATTER 和 MPI_GATHER 相似，但是分送和來自各個 CPU 的資料不必等長。

4.3 節 介紹 MPI_PACK 和 MPI_UNPACK 兩個資料集結與分解指令，以及同步指令 MPI_BARRIER 和取得時鐘時刻指令 MPI_WTIME。

4.4 節 說明使用這些 MPI 指令來把循序程式 T4SEQ 改寫成平行程式 T4DCP。

4.1 格點數不能整除的循序程式 T4SEQ

循序程式 T4SEQ 裏陣列的 dimension 是 161，只能夠被 7 和 23 整除。該程式除了陣列資料 A、B、C、D 之外，還用到三個純量資料 (scalar data) P、Q、R。在設定各個變數的初值 (initial value) 之後，也寫入磁檔，便利平行化時的驗證。

```
      PROGRAM T4SEQ
C
C     odd-dimensioned array with -1, +1 data reference
C
      PARAMETER (NTOTAL=161)
      REAL*8    A(NTOTAL), B(NTOTAL), C(NTOTAL), D(NTOTAL), P, Q, R
      DATA      P, Q, R/1.45, 2.62, 0.5/
C
C     Data generation
C
      DO I=1,NTOTAL
        B(I)=3.D0/DFLOAT(I)+1.D0
        C(I)=2.D0/DFLOAT(I)+1.D0
        D(I)=1.D0/DFLOAT(I)+1.D0
      ENDDO
      OPEN(1,FILE='input.dat',FORM='UNFORMATTED')
      WRITE(1) B
      WRITE(1) C
      WRITE(1) D
      WRITE(1) P,Q,R
      CLOSE(1)
C
C     Read input data & distribute input data
C
      OPEN(1,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
      READ (1) B
      READ (1) C
      READ (1) D
```

```
      READ (1) P,Q,R
C
C       Compute & output computed result
C
      DO I=2,NTOTAL-1
         A(I)=C(I)*D(I)*P+(B(I-1)+2.0*B(I)+B(I+1))*Q+R
      ENDDO
      WRITE(*,101) (A(I),I=1,NTOTAL,5)
101   FORMAT(10F8.3)
      STOP
      END
```

格點數不能整除的循序程式 T4SEQ 的測試結果如下 ：

```
    .000   18.550   15.720   14.682   14.143   13.812   13.588   13.427   13.305   13.210
  13.133   13.070   13.018   12.973   12.935   12.901   12.872   12.847   12.824   12.803
  12.785   12.768   12.753   12.739   12.726   12.714   12.703   12.693   12.684   12.675
  12.667   12.660    .000
```

4.2. MPI_SCATTERV、MPI_GATHERV

如果要把 4.1 節循序程式 T4SEQ 平行化，而且陣列也要切割時，其 dimension 是 161，只能夠被 7 和 23 整除。不能夠被 2、4、6、8 等參與平行計算的 CPU 數目整除，因此在 2、4、6、8 個 CPU 上平行計算時，就不能夠使用 MPI_SCATTER 來分送輸入資料，及使用 MPI_GATHER 來收集各個 CPU 上的資料。當然，此時還是可以使用 MPI_SEND 和 MPI_RECV 來完成陣列資料的分送與與收集。不過，MPI 還備有 MPI_SCATTERV 和 MPI_GATHERV 可以用來分送與與收集不等長陣列資料。

MPI_SCATTERV 和 MPI_SCATTER 的功能相似，MPI_GATHERV 和 MPI_GATHER 的功能相似，但是 MPI_SCATTER 是用來分送等量資料給每一個 CPU，MPI_GATHER 是收集來自每一個 CPU 的等量資料，MPI_SCATTERV 和 MPI_GATHERV 則不受這種限制。

MPI_SCATTERV 的叫用格式如下 ：

```
 CALL MPI_SCATTERV( T, GCOUNT, GDISP, MPI_REAL8,
1                   C(1), MYCOUNT, MPI_REAL8,
2                   IROOT, MPI_COMM_WORLD, IERR)
```

MPI_SCATTERV 是 IROOT CPU 把一個陣列 T 依 CPU id 的順序分段分送給每一個 CPU，包括 IROOT CPU 在內。由於分送給各個 CPU 的資料量可以不一樣多，因此必須使用一個陣列 GCOUNT 來存放分送給各個 CPU 的資料數量，再另外使用一個陣列 GDISP 來存放送出資料在 T 陣列上的相對位置。其引數依序為 ：

T	是待送出的資料陣列起點
GCOUNT	整數陣列，是存放要送給各個 CPU 的資料數量
GDISP	整數陣列，是存放要送給各個 CPU 資料起點在 T 陣列上的相對位置
MPI_REAL8	是待送出資料的類別
C(1)	是接收資料存放的起點
MYKOUNT	是接收資料的數量
MPI_REAL8	是接收資料的類別
IROOT	是送出資料陣的 CPU id

此處 GCOUNT 及 GDISP 是叫用 STARTEND 副程式算出來各個 CPU 上經切割後陣列片段的長度及各該陣列片段在未切割前的起點 START index 再減一。為了配合 CPU id 是從零起算，GCOUNT、GDISP 的 dimension 也是從零開始。所以 :

```
 REAL*8     A(0:N+1), B(0:N+1), C(0:N+1), D(0:N+1), T(NTOTAL)

 INTEGER    NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
1           L_NBR, R_NBR,
2           GSTART(0:31), GEND(0:31), GCOUNT(0:31), GDISP(0:31)

 CALL STARTEND (NPROC, 1, NTOTAL, GSTART, GEND, GCOUNT)
 DO I=0,NPROC-1
    GDISP(I)=GSTART(I)-1
 ENDDO
 MYOUNT= GCOUNT (MYID)
```

另外，分割後的陣列 A、B、C、D 的 dimension 是從零開始，而讀入資料是從一開始存放，

所以叫用 MPI_SCATTERV 時要標明從一開始 B(1)、C(1)、D(1)。還有 T 陣列上的相對位置 (displacement 或 offset) 是從零起算而不是從一起算，所以其值由 STARTEND 副程式算出來的 GSTART 要再減掉一。

MPI_GATHERV 的叫用格式如下 ：

```
      CALL MPI_GATHERV (A(1), MYKOUNT, MPI_REAL8,
     1                  T, GCOUNT, GDISP, MPI_REAL8,
     2                  IROOT, MPI_COMM_WORLD, IERR)
```

MPI_GATHERV 與 MPI_SCATTERV 的動作剛好相反，是 IROOT CPU 收集每一個 CPU (包括 IROOT CPU) 送給它的陣列 A 依 CPU id 的順序存入陣列 T 裏頭。由於來自各個 CPU 的資料量不一定一樣多，因此必須用一個陣列 GCOUNT 來存放來自各個 CPU 的資料數量，用另外一個陣列 GDISP 來存放要存入陣列 T 上的相對位置。其引數依序為 ：

A(1)	是待送出的資料陣列起點
MYKOUNT	是待送出資料的數量
MPI_REAL8	是待送出資料的類別
T	是接收資料存放的陣列位址
GCOUNT	整數陣列，是存放來自各個 CPU 的資料數量
GDISP	整數陣列，是存放來自各個 CPU 資料要存入 T 陣列的相對位置
MPI_REAL8	是接收資料的類別

IROOT 是接收資料的 CPU id

4.3 MPI_PACK、UNPACK、BARRIER、WTIME

循序程式 T4SEQ 裏除了陣列資料 A、B、C、D 之外，還用到三個純量資料 P、Q、R。在陣列資料切割與平行化之後，可以使用 MPI_SCATTERV 來分送輸入之陣列資料，使用 MPI_GATHERV 來收集由各個 CPU 計算出來的結果。至於三個純量資料 P、Q、R 在讀入之後，可以使用三個 MPI_BCAST 分別傳送給各個 CPU。不過，到現在為止，CPU 之間的資料傳送速度，比記憶體內資料移動的速度要慢上很多倍。所以在 CPU 之間一次傳送大量資料，比分多次傳送而每次只傳送少量資料來得有效率。

但是只有陣列資料才可以一次傳送多個資料，還好 MPI 備有資料集結副程式 MPI_PACK 可以把非連續位址資料 (noncontiguous data) 移入記憶體的連續位址裏 (contiguous memory locations)，俗稱緩衝區 (buffer area)，緩衝區為一個字符陣列 (character array)。然後把該字符陣列傳送給對方，對方收到該字符陣列後，再依存入的順序使用 MPI_UNPACK 取出各個變數的數值。

另外，還可以叫用 '同步' 副程式 MPI_BARRIER 使各個 CPU 達到同步的效果。叫用 MPI 涵數 (Fortran function) MPI_WTIME 來取得電腦時鐘時刻 (wall clock time)。

4.3.1 MPI_PACK、MPI_UNPACK

MPI_PACK 是要把非連續位址資料，像循序程式 T4SEQ 裏數個純量資料 P、Q、R 或數個陣列片段移入一個字符陣列裏。這些資料可以是同一種資料類別，也可以是不同的資料類別。

送出資料的 CPU 把集結 (pack) 好了的字符陣列傳送給對方，對方收到該字符陣列後，再依存入的順序使用 MPI_UNPACK 取出各個變數的數值。

要使用 MPI_PACK、MPI_UNPACK，首先要在該程式裏設定一個字符陣列。例如要存放三個倍準數 P、Q、R 的緩衝區，一個倍準數需要 8 個字符，三個倍準就需要 24 個字符。所以設定一個字符陣列 BUF1 其長度為 24 個字符。如下 ：

```
INTEGER       BUFSIZE
PARAMETER (BUFSIZE=24)
CHARACTER BUF1(BUFSIZE)
```

MPI_PACK 的叫用格式如下 ：

```
CALL MPI_PACK (P, 1, MPI_REAL8, BUF1, BUFSIZE, IPOS,
&                    MPI_COMM_WORLD, IERR)
```

其中引數 P 是要移入 BUF1 的變數起點

1 是要移入 BUF1 的資料數量

MPI_REAL8 是要移入 BUF1 的資料類別

BUF1 是存放集結資料的字符陣列起點

BUFSIZE 是字符陣列 BUF1 的字符長度

IPOS 是要存入 BUF1 的字符位置，從零開始，自動更新

IPOS 在移入第一筆資料之前要設為零，每次移入一筆資料，其值會自動更新，指向下一個

空位。所以 CPU0 讀入 P、Q、R 三個倍準數之後，再把他們移入 BUF1 可以寫成 ：

```
 IF (MYID.EQ.0) THEN
    READ(7) P,Q,R
    IPOS = 0
    CALL MPI_PACK (P, 1, MPI_REAL8, BUF1, BUFSIZE, IPOS,
1                       MPI_COMM_WORLD, IERR)
    CALL MPI_PACK (Q, 1, MPI_REAL8, BUF1, BUFSIZE, IPOS,
1                       MPI_COMM_WORLD, IERR)
    CALL MPI_PACK (R, 1, MPI_REAL8, BUF1, BUFSIZE, IPOS,
1                       MPI_COMM_WORLD, IERR)
 ENDIF
```

然後可以使用 MPI_BCAST 將字符陣列 BUF1 傳送給其他每一個 CPU 如下 ：

```
 IROOT=0
 CALL MPI_BCAST (BUF1, BUFSIZE, MPI_CHARACTER, IROOT,
&                       MPI_COMM_WORLD, IERR)
```

MPI_UNPACK 的叫用格式如下 ：

```
 CALL MPI_UNPACK (BUF1, BUFSIZE, IPOS, P, 1, MPI_REAL8,
&                        MPI_COMM_WORLD, IERR)
```

其中引數 BUF1　　是存放集結資料的字符陣列起點

BUFSIZE　是字符陣列 BUF1 的字符長度

IPOS　　是要從 BUF1 取出資料的字符位置，從零開始，自動更新

P　　是自 BUF1 取出的變數位址

1　　是自 BUF1 取出的資料數量

MPI_REAL8　　是自 BUF1 取出的資料類別

IPOS 在取出第一筆資料之前要設為零，每次取出一筆資料之後，其值會自動更新，指向下一筆資料的位置。所以各個 CPU 在執行過 bcast BUF1 之後，要自 BUF1 取出 P、Q、R 三個倍準數可以寫成 ：

```
 IF (MYID.NE.0) THEN
    IPOS=0
    CALL MPI_UNPACK (BUF1, BUFSIZE, IPOS, P, 1, MPI_REAL8,
&                          MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK (BUF1, BUFSIZE, IPOS, Q, 1, MPI_REAL8,
&                          MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK (BUF1, BUFSIZE, IPOS, R, 1, MPI_REAL8,
&                          MPI_COMM_WORLD, IERR)
 ENDIF
```

如果要 PACK 在一起的資料是同一種資料類別時，可以設定一個陣列，將這些資料逐一移入該陣列，然後將該陣列傳送給對方。對方收到該陣列後，再依序自該陣列移出各個資料。

```
 REAL*8   BUF1(3)

 IF (MYID.EQ.0) THEN
    READ(7) P,Q,R
    BUF1(1)=P
    BUF1(2)=Q
    BUF1(3)=R
 ENDIF
 IROOT=0
 CALL MPI_BCAST (BUF1, 3, MPI_REAL8, IROOT,
&                       MPI_COMM_WORLD, IERR)
 IF(MYID.NE.0) THEN
```

```
    P=BUF1(1)
    Q=BUF1(2)
    R=BUF1(3)
  ENDIF
```

4.3.2 MPI_BARRIER、MPI_WTIME

MPI_BARRIER 是屬於 '集體通訊' 類副程式，提供屬於同一個 communicator 的所有 CPU 達到 '同步' (synchronized) 的狀況。也就是每一個 CPU 在執行過 CALL MPI_BARRIER 之後，要等到所有 CPU 全部都執行過 CALL MPI_BARRIER 指令之後，才繼續執行 CALL MPI_BARRIER 之後的陳述。

MPI_BARRIER 的叫用格式如下 ：

```
CALL MPI_BARRIER (MPI_COMM_WORLD, IERR)
```

如果想要知道平行程式執行時的時鐘時刻 (wall clock time)，可利用 MPI 涵數 MPI_WTIME，其計量單位是 '秒鐘'。例如 ：

```
TIME1=MPI_WTIME()
```

其中 TIME1 的資料類別必須是 REAL*8，否則取得的時刻會產生錯誤。

如果想要知道參與平行計算各個 CPU 花費多少時間來完成平行程式的執行工作，可以在叫用 MPI_INIT 之後取得一個 MPI_WTIME，然後在叫用 MPI_FINALIZE 之前再取得一個 MPI_WTIME，後者減掉前者，就得到兩者之間所經歷的時間，例如 ：

```
CALL MPI_INIT(MPIERR)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
CALL MPI_BARRIER    (MPI_COMM_WORLD, IERR)
TIME1=MPI_WTIME()
. . .
TIME2=MPI_WTIME() – TIME1
PRINT *,' MYID, CLOCK TIME=', MYID, TIME2
CALL MPI_FINALIZE(MPI_ERR)
STOP
END
```

為甚麼在取得 TIME1 之前要執行 MPI_BARRIER 呢 ？ 因為 Job Scheduler 在執行平行程式之前，先要把該程式的執行檔 (executable file) 載入各個 CPU，這種載入多顆 CPU 的動作不可能完全同步。所以每一個 CPU 開始執行的時刻就有差異，但是平行程式裏一有資料傳輸，不管是點對點通訊或集體通訊，先到的就要等後到的，具有部份同步作用。而像範例程式這種小程式所需要的執行時間很短，一個 CPU 開始執行的時刻對該程式耗時統計就有很大的影響。有時可能發現某個 CPU 耗用的時間是其他 CPU 的數倍。於 TIME1 之前執行過 MPI_BARRIER 之後才開始計時，上述不合理現相就消失掉了。

同樣的道理，一個程式段落的耗時或資料傳送耗用多少時間等，都可以利用這一種方式來取得，然後加以累加即可。

4.4 資料切割的平行程式 T4DCP

平行程式 T4DCP 是將循序程式 T4SEQ 經過資料切割後的平行程式。未切割前陣列 A、B、C、D 的長度 NTOTAL 是 161，如果要在 4 個 CPU 上平行計算時，叫用副程式 STARTEND 算出各個 CPU 上陣列的長度分別為 41、40、40、40。所以在 NP 個 CPU 上平行計算時，切割後的陣列長度 N 為 NTOTAL / NP + 1。使用 PARAMETER 陳述設定如下 ：

```
PARAMETER (NTOTAL=161, NP=4, N=NTOTAL/NP+1)
```

再加上前後各保留一個轄區外陣列元素的位置，則所需的陣列長度為 N+2，其 DIMENSION 採用 (0:N+1) 的方式來撰寫，其轄區資料是存放在 index 1 到 N。於是陣列長度的設定為 ：

```
REAL*8     A(0:N+1),B(0:N+1),C(0:N+1),D(0:N+1), T(NTOTAL)
```

NTOTAL 不能為 NP 整除時，該 CPU 切割後的陣列長度可能是 N(=41) 或 N-1(=40)，各個 CPU 上陣列長度經過 STARTEND 算過後存放在陣列 GCOUNT 裏，其在 T 陣列上的相對位置是存放在陣列 GDISP 裏 ：

```
INTEGER GSTART(0:NP), GEND(0:NP), GCOUNT(0:NP), GDISP(0:NP)

CALL STARTEND (ID, NPROC, 1, NTOTAL, GSTART, GEND,GCOUNT)
```

所以該 CPU 切割後的陣列長度 MYKOUNT 及陣列的起迄 index ISTART、IEND 為 ：

```
MYKOUNT=GCOUNT(MYID)
ISTART=1
```

IEND=MYKOUNT

各個 CPU 切割後的陣列如圖 4.1 所示：

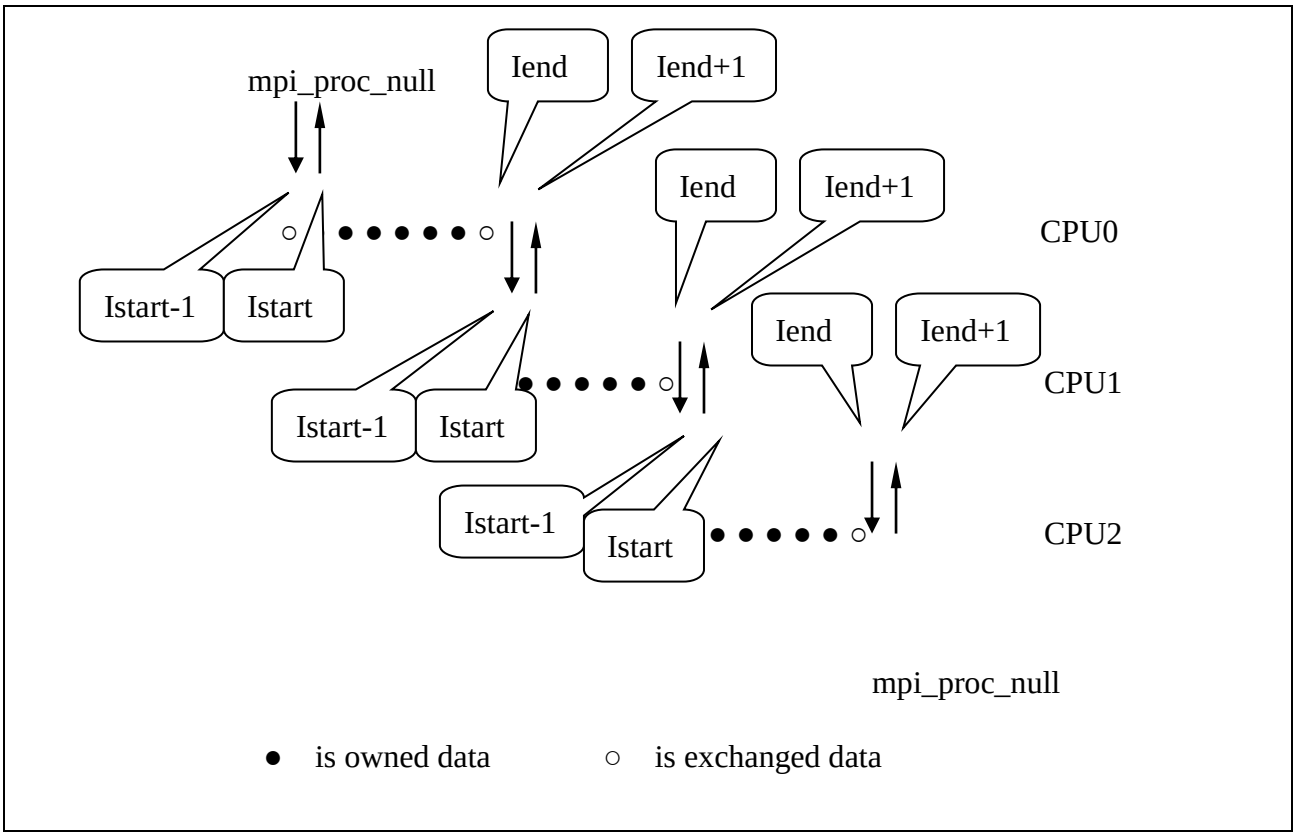


圖 4.1 格點數不能整除的資料交換平行計算示意圖

未切割前的 DO loop 為:

```
      DO I=2,NTOTAL-1
        A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
      ENDDO
```

則切割後 DO loop 的起迄 index 為 ：

```
      ISTART2=ISTART
      IF (MYID.EQ.0) ISTART2=2
      IEND1=IEND
      IF (MYID.EQ.NPROC-1) IEND1=IEND-1
```

格點數不能整除而且資料切割的平行程式 T4DCP 如下 ：

```
      PROGRAM T4DCP
C
C     Data partition with -1,+1 data exchange
C     using MPI_SCATTERV, MPI_GATHERV.
C     Each processor must hold at least 3 grid points.
C
      PARAMETER (NTOTAL=161, NP=4, N=NTOTAL/NP+1)
      INCLUDE    'mpif.h'
      REAL*8     A(0:N+1), B(0:N+1), C(0:N+1), D(0:N+1), T(NTOTAL),TIME1,TIME2

      INTEGER    NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1           L_NBR, R_NBR,
     2           GSTART(0:NP), GEND(0:NP), GCOUNT(0:NP), GDISP(0:NP)

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL MPI_BARRIER    (MPI_COMM_WORLD, IERR)
      TIME1=MPI_WTIME()
```

```
      CALL STARTEND (NPROC,1,NTOTAL,GSTART,GEND, GCOUNT)
      DO I=0,NPROC-1
          GDISP(I)=GSTART(I)-1
      ENDDO
      MYKOUNT=GCOUNT(MYID)
      PRINT *,' NPROC,MYID,ISTART,IEND,ARRAY LENGTH=', NPROC,MYID,
     &            GDISP(MYID)+1,GEND(MYID),MYCOUNT

      ISTART=1
      IEND= MYKOUNT
      ISTART2=ISTART
      IF(MYID.EQ.0) ISTART2=2
      IEND1=IEND
      IF(MYID.EQ.NPROC-1) IEND1=IEND-1

      ISTARTM1=0
      IENDP1=IEND+1
C
      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)           L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.NPROC-1) R_NBR=MPI_PROC_NULL

C
C     Read input data and SCATTER each array to each processor
C
      IF(MYID.EQ.0) THEN
        OPEN(7,FILE='input.dat',STATUS='OLD',FORM='UNFORMATTED')
        READ (7) T                    ! read array B
      ENDIF
      IROOT=0
      CALL MPI_SCATTERV (T, GCOUNT, GDISP, MPI_REAL8, B(1), MYKOUNT,
     1                   MPI_REAL8, IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
        READ (7) T                    ! read array C
      ENDIF
      CALL MPI_SCATTERV(T, GCOUNT, GDISP, MPI_REAL8, C(1), MYKOUNT,
     1                   MPI_REAL8, IROOT, MPI_COMM_WORLD, IERR)
```

```
      IF(MYID.EQ.0) THEN
        READ (7) T                    ! read array D
      ENDIF
      CALL MPI_SCATTERV (T, GCOUNT, GDISP, MPI_REAL8, D(1), MYKOUNT,
     1                   MPI_REAL8, IROOT, MPI_COMM_WORLD, IERR)
C
C     Exchange boundary data
C
      ITAG=110
      CALL MPI_SENDRECV (B(IEND),      1, MPI_REAL8, RIGHT_NBR, ITAG,
     1                   B(ISTARTM1), 1, MPI_REAL8,  LEFT_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
      ITAG=120
      CALL MPI_SENDRECV (B(ISTART), 1, MPI_REAL8,  LEFT_NBR, ITAG,
     1                   B(IENDP1), 1, MPI_REAL8, RIGHT_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
C
C     Compute, GATHER computed result & write out
C
CCC   DO I=2,NTOTAL-1
      DO I=ISTART2,IEND1
        A(I)=C(I)*D(I)+(B(I-1)+2.0*B(I)+B(I+1))*0.25
      ENDDO
      CALL MPI_GATHERV (A(1), MYKOUNT, MPI_REAL8,
     1                  T, GCOUNT, GDISP, MPI_REAL8,
     2                  IROOT, MPI_COMM_WORLD, IERR)
      IF(MYID.EQ.0) THEN
        WRITE(*,101) (T(I),I=1,NTOTAL,5)
      ENDIF
  101 FORMAT(10F8.3)
      TIME2=MPI_WTIME()-TIME1
      PRINT *,' MYID, CLOCK TIME=', MYID,TIME2
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

格點數不能整除的平行程式 T4DCP 的測試結果如下 :

```
TTENTION: 0031-408    4 nodes allocated by LoadLeveler, continuing...
   NPROC,MYID,ISTART,IEND,ARRAY LENGTH= 4    0      1      41    41
   NPROC,MYID,ISTART,IEND ARRAY LENGTH = 4    2    82    121    40
   NPROC,MYID,ISTART,IEND ARRAY LENGTH = 4    1    42     81    40
   NPROC,MYID,ISTART,IEND ARRAY LENGTH = 4    3    122    161    40
   MYID, CLOCK TIME= 2    0.8412782848E-01
   MYID, CLOCK TIME= 1    0.8406087756E-01
   MYID, CLOCK TIME= 3    0.8399387449E-01
      .000    18.550    15.720    14.682    14.143    13.812    13.588    13.427    13.305    13.210
   13.133    13.070    13.018    12.973    12.935    12.901    12.872    12.847    12.824    12.803
   12.785    12.768    12.753    12.739    12.726    12.714    12.703    12.693    12.684    12.675
   12.667    12.660       .000
   MYID, CLOCK TIME= 0    0.1014785469
```

從各個 CPU 列印出來的時間可以看出 CPU1、CPU2、CPU3 所耗用的時間相差無幾，而 CPU0 耗用較多。這是合理的，因為它比其他 CPU 多作了一些工作，像輸入資料的讀入與輸出資料的寫出等。

第五章　多維陣列的平行程式

前幾章所介紹的平行程式都是一維陣列的切割，這一章將討論多維陣列的切割。

5.1 節 說明循序程式 T5SEQ，該程式是由一個副程式改寫而成，與真實程式比較接近。它用到多個二維陣列與三維陣列，陣列的 index 順序也由原來的 (I,J,K) 改寫為 (K,J,I)。

5.2 節 說明在最後一維作計算切割而資料不切割的平行程式 T5CP。

5.3 節 說明末維計算及資料都切割的平行程式 T5DCP。

5.4 節 介紹與二維切割有關的 MPI 副程式。

5.5 節 介紹末二維切割的平行程式 T5_2D。

5.1 多維陣列的循序程式 T5SEQ

循序程式 T5SEQ 用到多個二維陣列與三維陣列，用到公用區域陣列 (COMMON array)，也用到局部陣列 (local array)。在計算之前的資料設定 (test data generation) 和計算之後的資料列印是為了驗證平行程式的正確與否。

```
      PROGRAM    T5SEQ
C
C       Sequential version of 2-D & 3-D array with -1,+1 data reference
C       I : U1,V1,PS1,HXU,HXV,HMMX,HMMY,VECINV,AM7
C       O : F1,F2
C       W : D7,D8,D00
C
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (KK=20, NN=120, MM=160, KM=3,MM1=MM-1,NN1=NN-1)

      DIMENSION    U1(KK,NN,MM),V1(KK,NN,MM),PS1(NN,MM)
      COMMON/BLK4/F1(KM,NN,MM),F2(KM,NN,MM),
     1                  HXU(NN,MM),HXV(NN,MM),HMMX(NN,MM),HMMY(NN,MM)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)

      DIMENSION     D7(NN,MM),D8(NN,MM),D00(KK,NN,MM)
C
C       Test data generation
C
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE(IC)/DBLE(IR)
      DO 10 I=1,MM1
      DO 10 J=1,NN
      DO 10 K=1,KK
        U1(K,J,I)=1.D0/DFLOAT(I)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   10 CONTINUE
      DO 20 I=1,MM
```

```
      DO 20 J=1,NN1
      DO 20 K=1,KK
        V1(K,J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   20 CONTINUE
      DO 30 I=1,MM
      DO 30 J=1,NN
        PS1(J,I)=1.D0/DFLOAT(I)+1.D0/DFLOAT(J)
        HXU(J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)
        HXV(J,I)=1.D0/DFLOAT(I)+2.D0/DFLOAT(J)
        HMMX(J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)
        HMMY(J,I)=1.D0/DFLOAT(I)+2.D0/DFLOAT(J)
   30 CONTINUE
      DO 40 K=1,KK
      AM7(K)=1.D0/DFLOAT(K)
      DO 40 KA=1,KK
        VECINV(KA,K)=1.D0/DFLOAT(KA)+1.D0/DFLOAT(K)
   40 CONTINUE
C
C       Start the computation
C
      DO 210 I=1,MM
      DO 210 J=1,NN
      DO 210 K=1,KM
        F1(K,J,I)=0.0D0
        F2(K,J,I)=0.0D0
  210 CONTINUE
      DO 220 I=1,MM1
      DO 220 J=2,NN1
        D7(J,I)=(PS1(J,I+1)+PS1(J,I))*0.5D0*HXU(J,I)
  220 CONTINUE
      DO 230 I=2,MM1
      DO 230 J=1,NN1
        D8(J,I)=(PS1(J+1,I)+PS1(J,I))*0.5D0*HXV(J,I)
  230 CONTINUE
      DO 240 I=2,MM1
      DO 240 J=2,NN1
      DO 240 K=1,KK
        D00(K,J,I)=(D7(J,I)*U1(K,J,I)-D7(J,I-1)*U1(K,J,I-1))*HMMX(J,I)
```

```
     &              +(D8(J,I)*V1(K,J,I)-D8(J-1,I)*V1(K,J-1,I))*HMMY(J,I)
 240 CONTINUE


      DO 260 I=2,MM1
      DO 260 KA=1,KK
      DO 260 J=2,NN1
      DO 260 K=1,KM
        F1(K,J,I)=F1(K,J,I)-VECINV(K,KA)*D00(KA,J,I)
 260 CONTINUE
      SUMF1=0.D0
      SUMF2=0.D0
      DO 270 I=2,MM1
      DO 270 J=2,NN1
      DO 270 K=1,KM
        F2(K,J,I)=-AM7(K)*PS1(J,I)
        SUMF1=SUMF1+F1(K,J,I)
        SUMF2=SUMF2+F2(K,J,I)
 270 CONTINUE
C
C       Output data for validation
C
      PRINT *,' SUMF1,SUMF2=', SUMF1,SUMF2
      PRINT *,' F2(2,2,I),I=1,160,5'
      PRINT 301,(F2(2,2,I),I=1,160,5)
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE(IC)/DBLE(IR)-CLOCK
      PRINT *,' CLOCK TIME=', CLOCK
      STOP
      END
```

多維陣列的循序程式 T5SEQ 的測試結果如下 :

```
 SUMF1,SUMF2= 26172.4605364288109    -2268.89180334316325
 F2(2,2,I),I=1,160,5
 .000E+00 -.333E+00 -.295E+00 -.281E+00 -.274E+00 -.269E+00 -.266E+00 -.264E+00
-.262E+00 -.261E+00 -.260E+00 -.259E+00 -.258E+00 -.258E+00 -.257E+00 -.257E+00
```

-.256E+00 -.256E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.254E+00
-.254E+00 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.253E+00 -.253E+00 -.253E+00
CLOCK TIME= 0.359999999996944098

5.2 多維陣列資料不切割的平行程式 T5CP

多維陣列資料切割後的資料傳輸效率，由於 Fortran 語言在記憶體上配置陣列元素時是採用 leading dimension 順序，也就是第一維 index 變動得最快而最後一維變動得最慢的順序，以切割最後一維的效率最好，否則就會產生不連續位址存取的現象，其傳輸效率較差。切割在最後一維時的切割方法和一維陣列的切割方法完全相同，唯一的差別是 index +1 或-1 的邊界資料交換數量在在一維陣列是一個點，在二維陣列是一條線，而三維陣列是一個面。

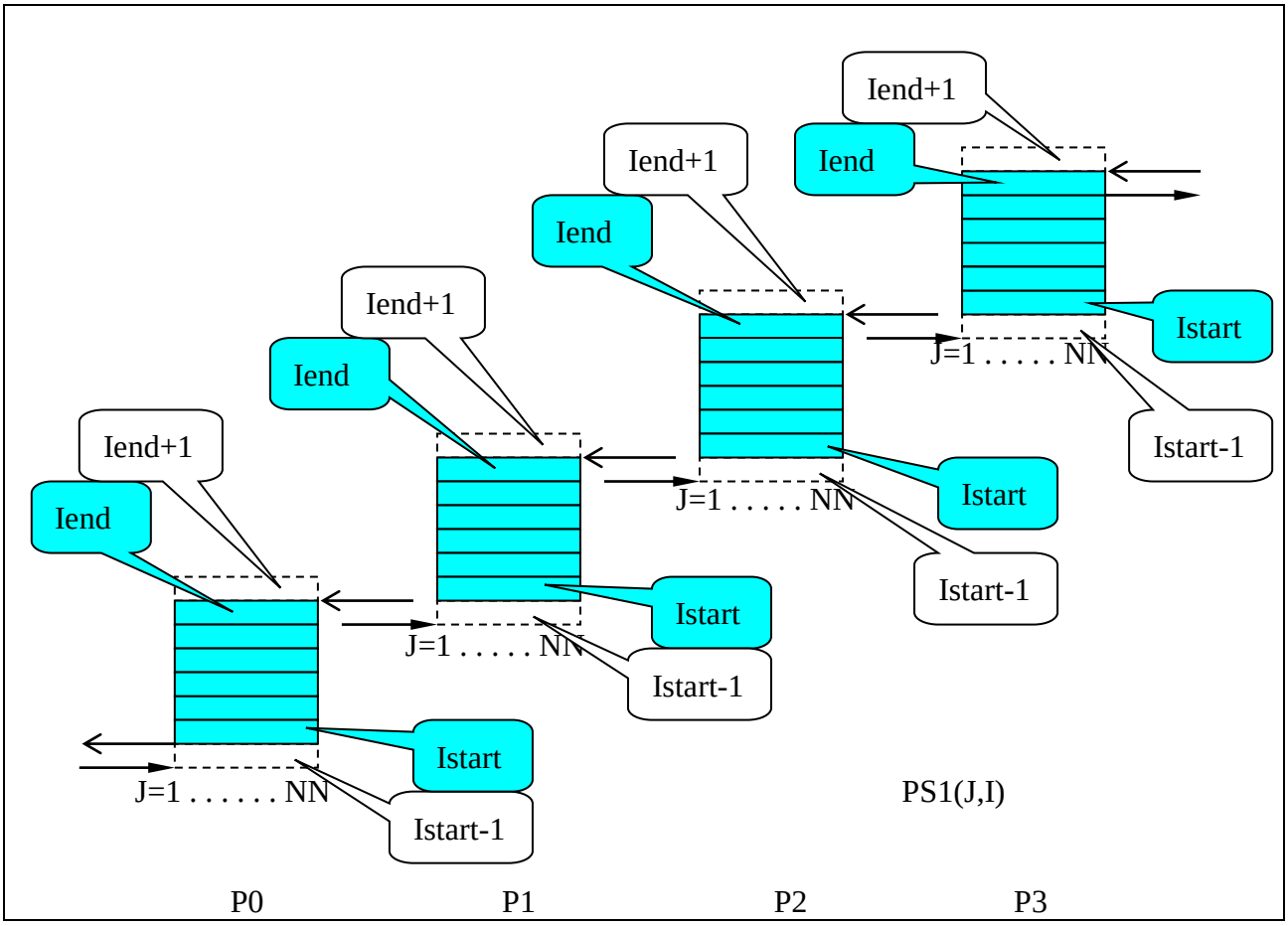


圖 5.1　二維陣列 PS1(NN,MM) 在第二維切割時的邊界資料交換

PARAMETER　(KK=20, NN=120, MM=160, KM=3, MM1=MM-1, NN1=NN-1)

```
 DIMENSION      U1(KK,NN,MM ),V1(KK,NN,MM ), PS1(NN,MM)

 ITAG=20
 CALL MPI_SENDRECV (PS1(1,ISTART), NN, MPI_REAL8, L_NBR, ITAG,
1                                  PS1(1,IENDP1), NN, MPI_REAL8, R_NBR, ITAG,
2                                  MPI_COMM_WORLD, ISTATUS, IERR)
 PARAMETER    (KK=20, NN=120, MM=160, KM=3, MM1=MM-1, NN1=NN-1)
 DIMENSION      U1(KK,NN,MM ),V1(KK,NN,MM ), PS1(NN,MM)

 NNKK=NN*KK
 ITAG=10
 CALL MPI_SENDRECV (U1(1,1,IEND),         NNKK, MPI_REAL8, R_NBR, ITAG,
1                                  U1(1,1,ISTARTM1), NNKK, MPI_REAL8, L_NBR, ITAG,
2                                  MPI_COMM_WORLD, ISTATUS, IERR)
```

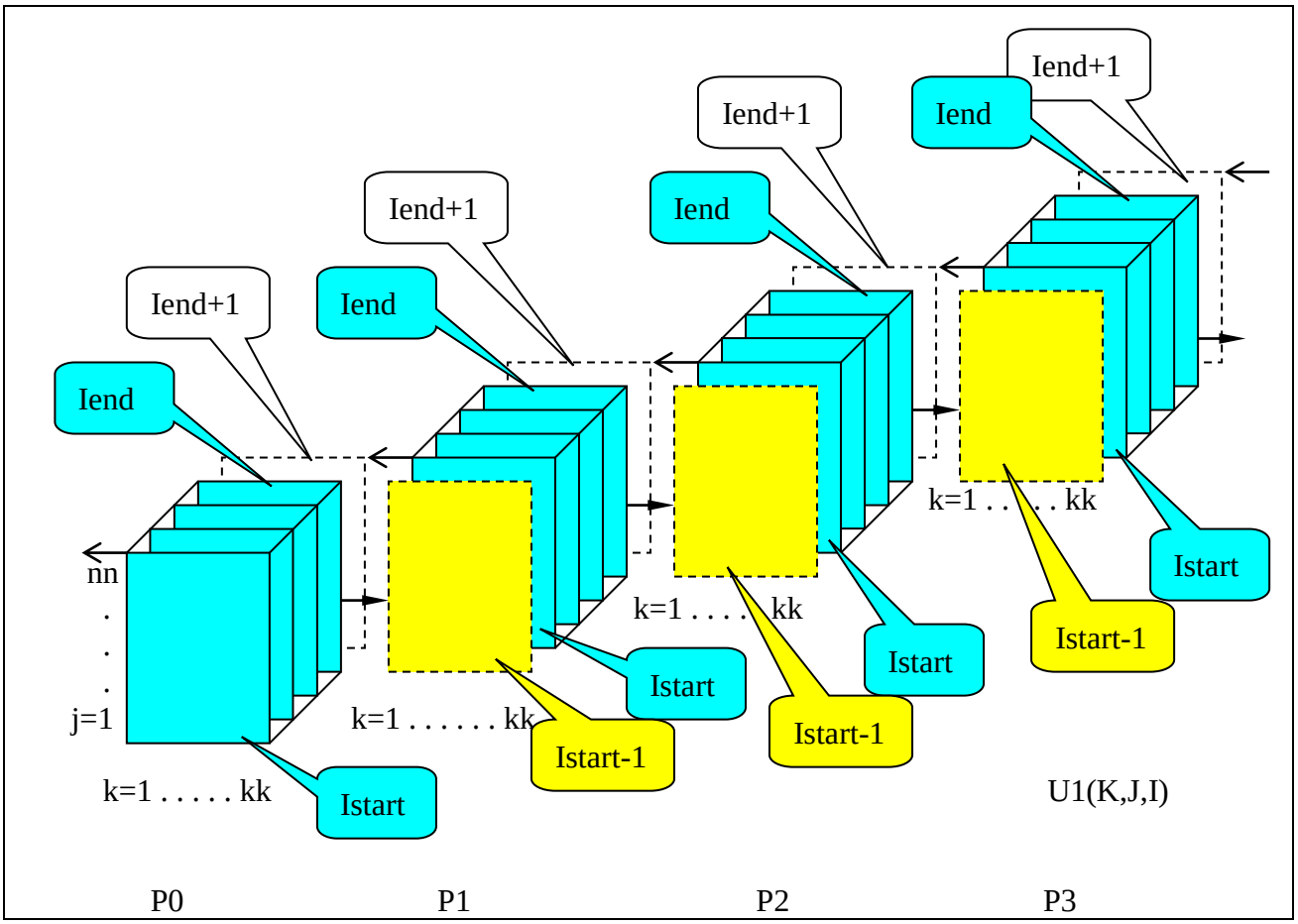


圖 5.2 三維陣列 U1(KK,NN,MM) 在第三維切割時的邊界資料交換

由一個主程式和數個副程式所組成的平行程式，可以將與 MPI 有關的變數放在一個公用區域 (COMMON block) 裏，像 NPROC、MYID、L_NBR、R_NBR、ISTART、IEND、ISTART2、IEND1、ISTARTM1、IENDP1 等等，放在 COMMON/BLKMPI/裏，其值在主程式設定一次就足夠，各個副程式裏只要具備相同的 COMMON block，就可以使用這些與 MPI 有關的變數。也可以把這些資料放在一個檔案裏，在各個主、副程式裏 INCLUDE 該檔案即可，像 INCLUDE 'mpif.h' 一樣。例如 ：

```
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C         mpi related data, this code is designed for up to 20 processors
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
          INTEGER            L_NBR, R_NBR, ISTATUS(MPI_STATUS_SIZE),
         1                    GCOUNT,GSTART,GEND
          COMMON/BLKMPI/MYID, NPROC, ISTATUS, L_NBR, R_NBR,
         1                       ISTART, ISTART2, ISTART3, IEND, IEND1,
         2                       ISTARTM1, IENDM1, IENDP1, MYCOUNT,
         3                       GCOUNT(0:31), GSTART(0:31), GEND(0:31)
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C         mpi related data will be set by main program, and pass them to
C         all subprograms by common block /BLKMPI/
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

多維陣列末維計算切割而資料不切割的平行程式 T5CP 如下 ：

```
          PROGRAM    T5CP
          IMPLICIT REAL*8 (A-H,O-Z)
          INCLUDE    'mpif.h'
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C         Parallel on the last dimension of multiple dimensional array
C         with -1,+1 data exchange without data partition
```

```
C         I : U1,V1,PS1,HXU,HXV,HMMX,HMMY,VECINV,AM7
C         O : F1,F2
C         W : D7,D8,D00
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
          PARAMETER    (KK=20, NN=120,MM=160, KM=3,MM1=MM-1,NN1=NN-1)

          DIMENSION    U1(KK,NN,MM ),V1(KK,NN,MM ),PS1(NN,MM)
          COMMON/BLK4/F1(KM,NN,MM),F2(KM,NN,MM),
         1                    HXU(NN,MM),HXV(NN,MM),HMMX(NN,MM),HMMY(NN,MM)
          COMMON/BLK5/VECINV(KK,KK),AM7(KK)

          DIMENSION      D7(NN,MM),D8(NN,MM),D00(KK,NN,MM)
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C         mpi related data, this code is designed for up to 20 processors
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
          INTEGER              L_NBR,R_NBR,ISTATUS(MPI_STATUS_SIZE),
         &                     GCOUNT,GSTART,GEND
          COMMON/BLKMPI/MYID,NPROC,ISTATUS,L_NBR,R_NBR,
         1                     ISTART,ISTART2,ISTART3, IEND,IEND1,
         2                     ISTARTM1, IENDM1,IENDP1, MYCOUNT,
         3                     GCOUNT(0:31),GSTART(0:31),GEND(0:31)
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C         mpi related data will be set by main program, and pass them to
C         all subprograms by common block /BLKMPI/
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
          CALL MPI_INIT (IERR)
          CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
          CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
          CALL MPI_BARRIER (MPI_COMM_WORLD, IERR)
          CLOCK=MPI_WTIME()
          CALL STARTEND (NPROC, 1, MM, GSTART, GEND, GCOUNT)
          ISTART=GSTART(MYID)
          IEND=GEND(MYID)
          print *,' MYID, NPROC, ISTART, IEND=',    MYID, NPROC, ISTART, IEND
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C**** Assume that each processor hold at least 3 grid points
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C               for DO I=x,MM1 (MM-1)
```

```
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      IEND1=IEND
      IF( MYID.EQ.NPROC-1) IEND1=IEND-1
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C           for DO I=2,x
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      ISTART2=ISTART
      IF( MYID.EQ.0) ISTART2=2
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C           for DO I=3,x
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      ISTART3=ISTART
      IF( MYID.EQ.0) ISTART3=3
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      ISTARTM1=ISTART-1
      IENDP1=IEND+1

      L_NBR=MYID-1
      R_NBR=MYID+1
      IF (MYID.EQ.0)              L_NBR=MPI_PROC_NULL
      IF (MYID.EQ.NPROC-1)    R_NBR=MPI_PROC_NULL
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C       Test data generation
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
CCC   DO 10 I=1,MM1
      DO 10 I=ISTART,IEND1
      DO 10 J=1,NN
      DO 10 K=1,KK
        U1(K,J,I)=1.D0/DFLOAT(I)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
 10   CONTINUE
CCC   DO 20 I=1,MM
      DO 20 I=ISTART,IEND
      DO 20 J=1,NN1
      DO 20 K=1,KK
        V1(K,J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
 20   CONTINUE
CCC   DO 30 I=1,MM
      DO 30 I=ISTART,IEND
```

```
      DO 30 J=1,NN
         PS1(J,I)=1.D0/DFLOAT(I)+1.D0/DFLOAT(J)
         HXU(J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)
         HXV(J,I)=1.D0/DFLOAT(I)+2.D0/DFLOAT(J)
         HMMX(J,I)=2.D0/DFLOAT(I)+1.D0/DFLOAT(J)
         HMMY(J,I)=1.D0/DFLOAT(I)+2.D0/DFLOAT(J)
   30 CONTINUE

      DO 40 K=1,KK
      AM7(K)=1.D0/DFLOAT(K)
      DO 40 KA=1,KK
         VECINV(KA,K)=1.D0/DFLOAT(KA)+1.D0/DFLOAT(K)
   40 CONTINUE
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
C       Start the computation
C-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      NNKK=NN*KK
      ITAG=10
      CALL MPI_SENDRECV (U1(1,1,IEND),     NNKK, MPI_REAL8, R_NBR, ITAG,
     1                   U1(1,1,ISTARTM1),NNKK,MPI_REAL8,L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
      ITAG=20
      CALL MPI_SENDRECV (PS1(1,ISTART), NN, MPI_REAL8, L_NBR,ITAG,
     1                   PS1(1,IENDP1), NN, MPI_REAL8, R_NBR,ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
CCC   DO 210 I=1,MM
      DO 210 I=ISTART,IEND
      DO 210 J=1,NN
      DO 210 K=1,KM
         F1(K,J,I)=0.0D0
         F2(K,J,I)=0.0D0
  210 CONTINUE

CCC   DO 220 I=1,MM1
      DO 220 I=ISTART,IEND1
      DO 220 J=2,NN1
         D7(J,I)=(PS1(J,I+1)+PS1(J,I))*0.5D0*HXU(J,I)
  220 CONTINUE
```

```
CCC   DO 230 I=2,MM1
      DO 230 I=ISTART2,IEND1
      DO 230 J=1,NN1
        D8(J,I)=(PS1(J+1,I)+PS1(J,I))*0.5D0*HXV(J,I)
 230  CONTINUE


      ITAG=30
      CALL MPI_SENDRECV(D7 (1,IEND),      NN, MPI_REAL8, R_NBR, ITAG,
     1                  D7(1,ISTARTM1), NN, MPI_REAL8, L_NBR, ITAG,
     2                  MPI_COMM_WORLD,ISTATUS,IERR)

CCC   DO 240 I=2,MM1
      DO 240 I=ISTART2,IEND1
      DO 240 J=2,NN1
      DO 240 K=1,KK
        D00(K,J,I)=(D7(J,I)*U1(K,J,I)-D7(J,I-1)*U1(K,J,I-1))*HMMX(J,I)
     1            +(D8(J,I)*V1(K,J,I)-D8(J-1,I)*V1(K,J-1,I))*HMMY(J,I)
 240  CONTINUE

CCC   DO 260 I=2,MM1
      DO 260 I=ISTART2,IEND1
      DO 260 KA=1,KK
      DO 260 J=2,NN1
      DO 260 K=1,KM
        F1(K,J,I)=F1(K,J,I)-VECINV(K,KA)*D00(KA,J,I)
 260  CONTINUE
      SUMF1=0.D0
      SUMF2=0.D0
CCC   DO 270 I=2,MM1
      DO 270 I=ISTART2,IEND1
      DO 270 J=2,NN1
      DO 270 K=1,KM
        F2(K,J,I)=-AM7(K)*PS1(J,I)
        SUMF1=SUMF1+F1(K,J,I)
        SUMF2=SUMF2+F2(K,J,I)
 270  CONTINUE
```

```
C-----------------------------------------------------------------------
C         Output data for validation
C-----------------------------------------------------------------------
        IROOT=0
        CALL MPI_REDUCE (SUMF1, GSUMF1, 1, MPI_REAL8, MPI_SUM, IROOT,
       1                          MPI_COMM_WORLD, IERR)
        CALL MPI_REDUCE (SUMF2, GSUMF2, 1, MPI_REAL8, MPI_SUM, IROOT,
       1                          MPI_COMM_WORLD, IERR)
        ITAG=40
        IF (MYID.NE.0) THEN
          KOUNT=KM*NN*MYCOUNT
          CALL MPI_SEND (F2(1,1,ISTART), KOUNT, MPI_REAL8, 0, ITAG,
       1                          MPI_COMM_WORLD, IERR)
        ELSE
          DO ISRC=1,NPROC-1
            ISTART1=GSTART(ISRC)
            KOUNT1=KM*NN*GCOUNT(ISRC)
            CALL MPI_RECV (F2(1,1,ISTART1), KOUNT1, MPI_REAL8, ISRC, ITAG,
       1                            MPI_COMM_WORLD, ISTATUS, IERR)
          ENDDO
        ENDIF
 301    FORMAT(8E10.3)
        IF(MYID.EQ.0) THEN
          PRINT *,'SUMF1,SUMF2=', GSUMF1,GSUMF2
          PRINT *,' F2(2,2,I),I=1,160,5'
          PRINT 301,(F2(2,2,I),I=1, 160,5)
        ENDIF
        CLOCK=MPI_WTIME() - CLOCK
        PRINT *,' MYID, CLOCK TIME=', MYID,CLOCK
        CALL MPI_FINALIZE(IERR)
        STOP
        END
```

多維陣列末維計算切割而資料不切割的平行程式 T5CP 的測試結果如下 :

```
ATTENTION: 0031-408  8 nodes allocated by LoadLeveler, continuing...
  MYID,NPROC,ISTART,IEND= 0 8 1 20
```

```
 MYID,NPROC,ISTART,IEND= 2 8 41 60
 MYID,NPROC,ISTART,IEND= 7 8 141 160
 MYID,NPROC,ISTART,IEND= 4 8 81 100
 MYID,NPROC,ISTART,IEND= 5 8 101 120
 MYID,NPROC,ISTART,IEND= 1 8 21 40
 MYID,NPROC,ISTART,IEND= 3 8 61 80
 MYID,NPROC,ISTART,IEND= 6 8 121 140
 MYID, CLOCK TIME= 1    0.661787500139325857E-01
 MYID, CLOCK TIME= 4    0.735177251044660807E-01
 MYID, CLOCK TIME= 2    0.681649250909686089E-01
SUMF1,SUMF2= 26172.4605364287745    -2268.89180334311050
 F2(2,2,I),I=1,160,5
 .000E+00 -.333E+00 -.295E+00 -.281E+00 -.274E+00 -.269E+00 -.266E+00 -.264E+00
-.262E+00 -.261E+00 -.260E+00 -.259E+00 -.258E+00 -.258E+00 -.257E+00 -.257E+00
-.256E+00 -.256E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.254E+00
-.254E+00 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.253E+00 -.253E+00 -.253E+00
 MYID, CLOCK TIME= 0    0.816429499536752701E-01
 MYID, CLOCK TIME= 3    0.701620501931756735E-01
 MYID, CLOCK TIME= 5    0.754272749181836843E-01
 MYID, CLOCK TIME= 6    0.774252500850707293E-01
 MYID, CLOCK TIME= 7    0.800943500362336636E-01
```

執行循序程式 T5SEQ 需時 0.36 秒，在八個 CPU 上執行平行程式 T5CP 需時 0.0816 秒，平行效率 (parallel speed up) 為 0.36/0.0816= 4.4 倍。

5.3 多維陣列末維資料切割的平行程式 T5DCP

平行程式 T5DCP 是將循序程式 T5SEQ 經過資料切割後的平行程式。其陣列資料只就最後一維作切割。未切割前最後一維的長度 MM 是 160，經副程式 STARTEND 算出在 NP 個 CPU 上平行計算時，MM 能被 NP 整除時切割後的陣列長度 M 為 MM/NP，MM 不能被 NP 整除時切割後的陣列長度 M 必須為 MM/NP+1。使用 PARAMETER 陳述設定如下 ：

```
      PARAMETER    (KK=20, NN=120, MM=160, KM=3, MM1=MM-1, NN1=NN-1)
      PARAMETER    (NP=8, M=MM/NP)
```

則需要做資料交換的 dimension 設為 (0:M+1)，不需要做資料交換的 dimension 設為 (M)。

```
      DIMENSION       TT(KM,NN,MM)
      DIMENSION       U1(KK,NN,0:M+1), V1(KK,NN,M), PS1(NN,0:M+1)
      COMMON/BLK4/F1(KM,NN,M), F2(KM,NN,M),
     1                HXU(NN,M), HXV(NN,M),
     2                HMMX(NN,M), HMMY(NN,M)
      COMMON/BLK5/VECINV(KK,KK), AM7(KK)
      DIMENSION       D7(NN,0:M+1), D8(NN,M), D00(KK,NN,M)
```

為了使這個程式能夠同時適應 MM 能被 NP 整除及 MM 不能被 NP 整除的情況，在收集計算結果 F1、F2 時使用 MPI_GATHERV 而不使用 MPI_GATHER。此時陣列 F1、F2 在各個 CPU 上的資料長度 GCOUNT 和在陣列 TT 上的相對位置 GDISP 為 ：

```
      CALL STARTEND (NPROC, 1, MM, GSTART, GEND,GCOUNT)
      DO I=0,NPROC-1
        GCOUNT(I)=KM*NN*GCOUNT(I)
        GDISP(I)=KM*NN*(GSTART(I)-1)
```

```
      ENDDO
      ISTARTG=GSTART(MYID)
      IENDG=GEND(MYID)
      MYCOUNT=IENDG-ISTARTG+1
```

三維陣列資料上的 MPI_GATHERV 寫法如下 ：

```
      IROOT=0
      KOUNT= GCOUNT(MYID)
      CALL MPI_GATHERV (F1, KOUNT, MPI_REAL8,
     1                  TT, GCOUNT, GDISP, MPI_REAL8,
     2                  IROOT, MPI_COMM_WORLD, IERR)
```

多維陣列末維資料切割的平行程式如下 ：

```
      PROGRAM   T5DCP
      IMPLICIT REAL*8 (A-H,O-Z)
      INCLUDE   'mpif.h'
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Data partition on the last dimension of muti-dimensional arrays
C       with -1,+1 data exchange
C       NP=8 MUST be modified when parallelized on other than 8 processors
C       I : U1,V1,PS1,HXU,HXV,HMMX,HMMY,VECINV,AM7
C       O : F1,F2
C       W : D7,D8,D00
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      PARAMETER   (KK=20,NN=120,MM=160, KM=3,MM1=MM-1,NN1=NN-1)
      PARAMETER   (NP=8, M=MM/NP+1)
      DIMENSION     TT(KM,NN,MM)
      DIMENSION     U1(KK,NN,0:M+1),V1(KK,NN,M),PS1(NN,0:M+1)
      COMMON/BLK4/F1(KM,NN,M),F2(KM,NN,M),
     1             HXU(NN,M),HXV(NN,M),
     2             HMMX(NN,M),HMMY(NN,M)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)
```

```
      DIMENSION     D7(NN,0:M+1),D8(NN,M),D00(KK,NN,M)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       mpi related data, this code is designed for up to NP processors
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      INTEGER              L_NBR, R_NBR, ISTATUS(MPI_STATUS_SIZE),
     &                     GSTART, GEND, GCOUNT, GDISP, MYCOUNT
      COMMON/BLKMPI/MYID, NPROC, ISTATUS, L_NBR, R_NBR,
     1                     ISTART, ISTART2, ISTART3, IEND, IEND1,
     2                     ISTARTM1, IENDM1, IENDP1, MYCOUNT ,
     3       GSTART(0:NP), GEND(0:NP), GCOUNT(0:NP), GDISP(0:NP)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       mpi related data will be set by main program, and pass them to
C       all subprograms by common block /BLKMPI/
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL MPI_BARRIER (MPI_COMM_WORLD, MPI_ERR)
      CLOCK=MPI_WTIME()
      CALL STARTEND(NPROC,1,MM,GSTART,GEND,GCOUNT)
      DO I=0,NPROC-1
        GCOUNT(I)=KM*NN*GCOUNT(I)
        GDISP(I)=KM*NN*(GSTART(I)-1)
      ENDDO
      ISTARTG=GSTART(MYID)
      IENDG=GEND(MYID)
      MYCOUNT =IENDG-ISTARTG+1
      PRINT *,' MYID,NPROC,ISTARTG,IENDG=', MYID,NPROC,ISTARTG,IENDG
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Assume that each processor hold at least 3 grid points
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      ISTART=1
      IEND=MYCOUNT
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C           for DO I=x,MM1 (MM-1)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      IEND1=IEND
      IF( MYID.EQ.NPROC-1) IEND1=IEND1-1
```

```
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C            for DO I=2,x
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
        ISTART2=1
        IF( MYID.EQ.0) ISTART2=2
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
        ISTARTM1=ISTART-1
        IENDP1=IEND+1

        L_NBR=MYID-1
        R_NBR=MYID+1
        IF(MYID.EQ.0)    L_NBR=MPI_PROC_NULL
        IF(MYID.EQ.NPROC-1)    R_NBR=MPI_PROC_NULL
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C        Test data generation
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
CCC     DO 10 I=1,MM1
        DO 10 I=1,IEND1
        II=I+ISTARTG-1
        DO 10 J=1,NN
        DO 10 K=1,KK
          U1(K,J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   10   CONTINUE
CCC     DO 20 I=1,MM
        DO 20 I=1,IEND
        II=I+ISTARTG-1
        DO 20 J=1,NN1
        DO 20 K=1,KK
          V1(K,J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   20   CONTINUE
CCC     DO 30 I=1,MM
        DO 30 I=1,IEND
        II=I+ISTARTG-1
        DO 30 J=1,NN
          PS1(J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(J)
          HXU(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)
          HXV(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(J)
          HMMX(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)
```

```
          HMMY(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(J)
   30   CONTINUE
        DO 40 K=1,KK
        AM7(K)=1.D0/DFLOAT(K)
        DO 40 KA=1,KK
          VECINV(KA,K)=1.D0/DFLOAT(KA)+1.D0/DFLOAT(K)
   40   CONTINUE
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Start the computation
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
        NNKK=NN*KK
        ITAG=10
        CALL MPI_SENDRECV (U1(1,1,IEND ), NNKK, MPI_REAL8, R_NBR, ITAG,
       1                   U1(1,1,ISTARTM1), NNKK, MPI_REAL8, L_NBR, ITAG,
       2                   MPI_COMM_WORLD, ISTATUS, IERR)
        ITAG=30
        CALL MPI_SENDRECV (PS1(1,ISTART), NN, MPI_REAL8, L_NBR, ITAG,
       1                   PS1(1,IENDP1), NN, MPI_REAL8, R_NBR, ITAG,
       2                   MPI_COMM_WORLD, ISTATUS, IERR)
CCC    DO 210 I=1,MM
        DO 210 I=ISTART,IEND
        DO 210 J=1,NN
        DO 210 K=1,KM
          F1(K,J,I)=0.0D0
          F2(K,J,I)=0.0D0
  210   CONTINUE

CCC    DO 220 I=1,MM1
        DO 220 I=ISTART,IEND1
        DO 220 J=2,NN1
          D7(J,I)=(PS1(J,I+1)+PS1(J,I))*0.5D0*HXU(J,I)
  220   CONTINUE

CCC    DO 230 I=2,MM1
        DO 230 I=ISTART2,IEND1
        DO 230 J=1,NN1
          D8(J,I)=(PS1(J+1,I)+PS1(J,I))*0.5D0*HXV(J,I)
  230   CONTINUE
```

```
      ITAG=50
      CALL MPI_SENDRECV (D7(1,IEND),        NN, MPI_REAL8, R_NBR, ITAG,
     1                   D7(1,ISTARTM1), NN, MPI_REAL8, L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
CCC   DO 240 I=2,MM1
      DO 240 I=ISTART2,IEND1
      DO 240 J=2,NN1
      DO 240 K=1,KK
        D00(K,J,I)=(D7(J,I)*U1(K,J,I)-D7(J,I-1)*U1(K,J,I-1))*HMMX(J,I)
     1             +(D8(J,I)*V1(K,J,I)-D8(J-1,I)*V1(K,J-1,I))*HMMY(J,I)
 240  CONTINUE

CCC   DO 260 I=2,MM1
      DO 260 I=ISTART2,IEND1
      DO 260 KA=1,KK
      DO 260 J=2,NN1
      DO 260 K=1,KM
        F1(K,J,I)=F1(K,J,I)-VECINV(K,KA)*D00(KA,J,I)
 260  CONTINUE

      SUMF1=0.D0
      SUMF2=0.D0
CCC   DO 270 I=2,MM1
      DO 270 I=ISTART2,IEND1
      DO 270 J=2,NN1
      DO 270 K=1,KM
        F2(K,J,I)=-AM7(K)*PS1(J,I)
        SUMF1=SUMF1+F1(K,J,I)
        SUMF2=SUMF2+F2(K,J,I)
 270  CONTINUE
C   - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Output data for validation
C   - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      IROOT=0
      CALL MPI_REDUCE (SUMF1, GSUMF1, 1, MPI_REAL8, MPI_SUM,
     1                 IROOT, MPI_COMM_WORLD, IERR)
```

```
      CALL MPI_REDUCE (SUMF2, GSUMF2, 1, MPI_REAL8, MPI_SUM,
     1                 IROOT, MPI_COMM_WORLD, IERR)
      KOUNT=GCOUNT(MYID)
      CALL MPI_GATHERV (F2, KOUNT, MPI_REAL8,
     1                  TT, GCOUNT, GDISP, MPI_REAL8,
     3                  IROOT, MPI_COMM_WORLD, IERR)

      IF (MYID.EQ.0) THEN
         PRINT *,'SUMF1,SUMF2=',    GSUMF1,GSUMF2
         PRINT *,' F2(2,2,I),I=1, 160,5'
         PRINT 301,(TT(2,2,I),I=1, 160,5)
      ENDIF
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,' MYID, CLOCK TIME=',    MYID,CLOCK
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

多維陣列末維資料切割的平行程式 T5DCP 的測試結果如下 ：

```
ATTENTION: 0031-408  8 nodes allocated by LoadLeveler, continuing...
   MYID,NPROC,ISTARTG,IENDG= 0  8  1  20
   MYID,NPROC,ISTARTG,IENDG= 7  8  141  160
   MYID,NPROC,ISTARTG,IENDG= 3  8  61  80
   MYID,NPROC,ISTARTG,IENDG= 6  8  121  140
   MYID,NPROC,ISTARTG,IENDG= 2  8  41  60
   MYID,NPROC,ISTARTG,IENDG= 1  8  21  40
   MYID,NPROC,ISTARTG,IENDG= 4  8  81  100
   MYID,NPROC,ISTARTG,IENDG= 5  8  101  120
   MYID, CLOCK TIME= 7  0.720097750891000032E-01
   MYID, CLOCK TIME= 3  0.719586000777781010E-01
   MYID, CLOCK TIME= 5  0.720054251141846180E-01
   MYID, CLOCK TIME= 1  0.698123250622302294E-01
SUMF1,SUMF2= 26172.4605364287745 -2268.89180334311050
   F2(2,2,I),I=1,160,5
   .000E+00 -.333E+00 -.295E+00 -.281E+00 -.274E+00 -.269E+00 -.266E+00 -.264E+00
```

```
 -.262E+00 -.261E+00 -.260E+00 -.259E+00 -.258E+00 -.258E+00 -.257E+00 -.257E+00
 -.256E+00 -.256E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.254E+00
 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.253E+00 -.253E+00 -.253E+00
   MYID, CLOCK TIME= 0    0.832411749288439751E-01
   MYID, CLOCK TIME= 2    0.763444001786410809E-01
   MYID, CLOCK TIME= 4    0.821004498284310102E-01
   MYID, CLOCK TIME= 6    0.828509500715881586E-01
```

執行循序程式 T5SEQ 需時 0.36 秒，在八個 CPU 上執行平行程式 T5DCP 需時 0.0832 秒，平行效率 (parallel speed up) 為 0.36/0.0832 = 4.33 倍。在八個 CPU 上平行計算時，採用陣列末維資料切割方法需時 0.0832，而採用資料不切割方法也是需時 0.082 秒，兩者的平行效率差別不大。

5.4 與二維切割有關的 MPI 副程式

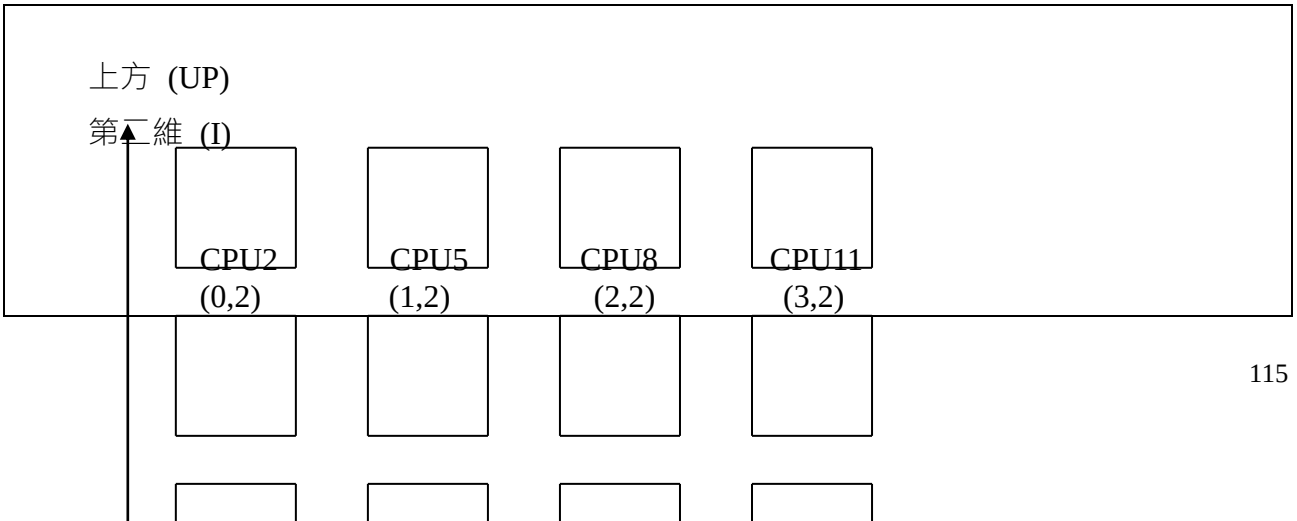
多維陣列的切割方式有許多種，5.3 節介紹最簡單的最後一維切割方式的程式寫法。此外，常用的切割方式還有二維切割與多維切割。這一節將先介紹二維切割有關的 MPI 副程式 MPI_CART_CREATE、MPI_CART_COORDS、MPI_CART_SHIFT、MPI_TYPE_VECTOR、MPI_TYPE_COMMIT。下一節再介紹使用這些副程式來撰寫二維切割的平行程式。

5.4.1 垂直座標圖示法則 (Cartesian Topology)

二維陣列 A(NN,MM) 要做二維切割時，便要敘明要在第一維和第二維各切成幾塊。例如第一維要切成四塊，第二維要切成三塊，則切割後第一維的長度 N 為 NN/4，第二維的長度 M 為 MM/3，當然兩者都必須能整除才行。如果 NN 和 MM 的值分別為 200 和 150，就可以使用 PARAMETER 來設定 M、N 的值，切割後陣列的第一維和第二維都要預留前後各一個陣列元素位置時，其 dimension 可設定為 ：

```
PARAMETER (NN=200, MM=150, JP=4, IP=3, N=NN/JP, M=MM/IP,)
DIMENSION    A(0:N+1, 0:M+1)
```

陣列 A 切成 4x3 十二塊，每一塊由一個 CPU 來執行時，就需要十二個 CPU。這十二個 CPU 的編號和座標如圖 5.3 所示，也就是切割後陣列 A 十二塊小陣列的編號和座標 ：



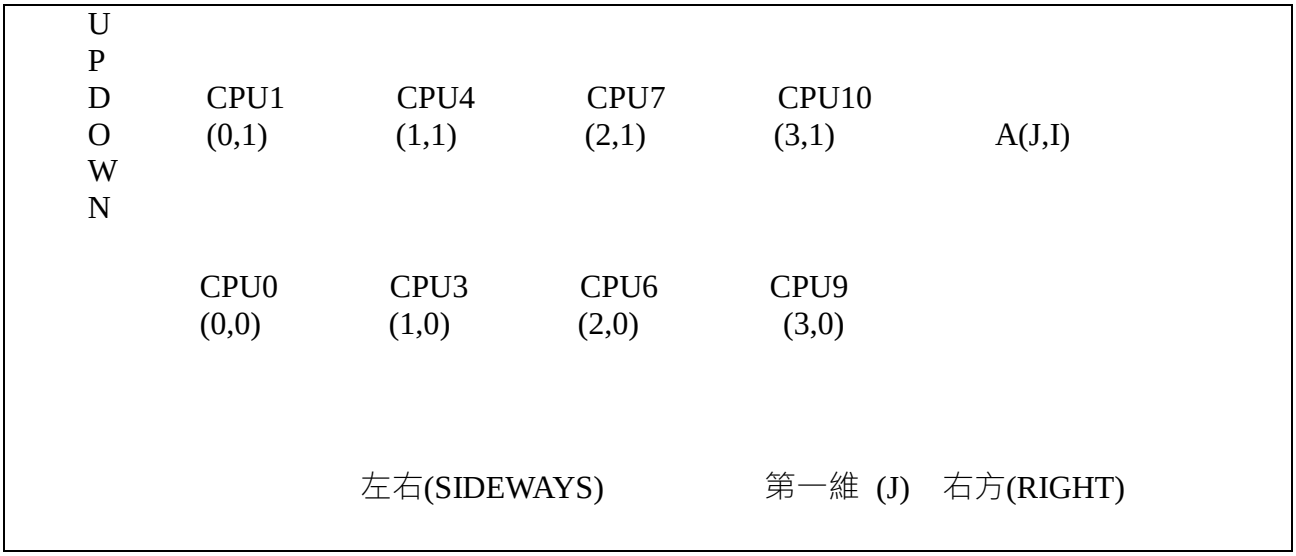


圖 5.3 二維切割區塊圖

十二個 CPU 之中每一個 CPU 的上、下、左、右 鄰居關係如圖 5.3 所示。水平方向 X 軸是第一維 (J)，垂直方向 Y 軸是第二維 (I)。

5.4.2 界定二維切割的 MPI 副程式 MPI_CART_CREATE、MPI_CART_COORDS、MPI_CART_SHIFT

二維切割方式必須在叫用 MPI_COMM_SIZE、MPI_COMM_RANK 分別取得 NPROC、MYID 等之後，再叫用 MPI 副程式 MPI_CART_CREATE 來加以設定。如圖 5.3 的二維切割方式，其相關引數設定如下 ：

```
PARAMETER (NDIM=2, JP=4, IP=3)
INTEGER NPROC,MYID
INTEGER IPART(NDIM), COMM2D, MY_CID, MY_COORD(NDIM)
INTEGER SIDEWAYS, UPDOWN, RIGHT, UP, L_NBR, R_NBR, T_NBR, B_NBR
LOGICAL PERIODS(NDIM), REORDER
. . .
CALL MPI_INIT (IERR)
```

```
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)

IPART(1)=JP
IPART(2)=IP
PERIODS(1)=.FALSE.
PERIODS(2)=.FALSE.
REORDER=.TRUE.

CALL MPI_CART_CREATE (MPI_COMM_WORLD, NDIM, IPART, PERIODS,
&                     REORDER, COMM2D, IERR)
```

引數

MPI_COMM_WORLD	是原來的 communicator
NDIM	是切割的維數，圖 5.3 二維切割的例子要設為 2
IPART	是 NDIM 個元素的整數陣列
IPART(1)	是第一維切成的塊數，圖 5.3 的例子要設為 4
IPART(2)	是第二維切成的塊數，圖 5.3 的例子要設為 3
PERIODS	是 NDIM 個元素的羅輯陣列
PERIODS(1)	是第一維首尾區塊是否相鄰，是為 .TRUE. 否為 .FALSE，圖 5.3 的例子要設為 .FALSE.
PERIODS(2)	是第二維首尾區塊是否相鄰，是為 .TRUE. 否為 .FALSE，圖 5.3 的例子要設為 .FALSE.
REORDER	是個羅輯變數，已排定的 CPU 要不要重新安排， 一般都設為 .TRUE.
COMM2D	是為這種切割方式新命名的 communicator

於是這個 4x3 區塊的二維切割 communicator 就叫做 COMM2D。

在叫用 MPI_CART_CREATE 之前已經叫用過 MPI_COMM_RANK，已經安排好各個 CPU 了，那是一種直線安排。在叫用 MPI_CART_CREATE 時，是一種平面安排，當 REORDER 的值設為 .TRUE. 時，允許電腦系統重新安排各個 CPU，使得相鄰的 CPU 安排在相鄰的位置，而得到最佳的傳輸效率。

往後，程式是在新的 communicator COMM2D 之下運作，原取得的 MYID 已經不適用。必須重新叫用 MPI_COMM_RANK 取得在新的 communicator COMM2D 裏的 CPU id。

MPI_COMM_RANK 的叫用格式如下 ：

CALL MPI_COMM_RANK (COMM2D, MYID, IERR)

引數 COMM2D 是新設定了的 communicator

MYID 是 communicator COMM2D 之下新的 CPU id

MYID 的安排方式如圖 5.3 的 CPU0、CPU1、CPU2 等所示。接下來就必須叫用 MPI_CART_COORDS 取得該 CPU 在二維 CPU 陣列裏的座標 MY_COORD。

MPI_CART_COORDS 的叫用格式如下 ：

CALL MPI_CART_COORDS (COMM2D, MY_CID, NDIM, MY_COORD, IERR)

引數 COMM2D 是新設定了的 communicator

MY_CID 是 communicator COMM2D 之下的 CPU id

NDIM　　是切割的維數，圖 5.3 的例子要設為 2

MY_COORD　是 NDIM 個元素的整數陣列，存放 MY_CID CPU 的 CPU 陣列座標，

MY_COORD(1) 是第一維方向的座標，從零起算

MY_COORD(2) 是第二維方向的座標，從零起算

CPU 座標 MY_COORD 的安排方式如圖 5.3 CPU0、CPU1、CPU2 等 CPU id 之下的括弧內數字。

由圖 5.3 可知，當 MY_COORD(1) 之值為零時，該 CPU 是在 CPU 陣列裏的最左邊。當 MY_COORD(1) 之值為 JP-1 時，該 CPU 是在 CPU 陣列裏的最右邊。當 MY_COORD(2) 之值為零時，該 CPU 是在 CPU 陣列裏的最下面 (底邊)，當 MY_COORD(2) 之值為 IP-1 時，該 CPU 是在 CPU 陣列裏的最上面 (頂邊)。

然後還必須叫用 MPI_CART_SHIFT 來取得該 CPU 的上、下、左、右鄰居的 CPU id。

MPI_CART_SHIFT 的叫用格式如下 ：

```
INTEGER    SIDEWAYS,UPDOWN,RIGHT,UP
SIDEWAYS=0
UPDOWN=1
RIGHT=1
UP=1

CALL MPI_CART_SHIFT (COMM2D, SIDEWAYS, RIGHT, L_NBR, R_NBR, IERR)
```

引數　COMM2D　　是新設定了的 communicator

SIDEWAYS　是個整數變數，其值設為零是要取得第一維 (J 方向) 的鄰居

RIGHT　是個整數變數，其值設為一是要取得左、右 鄰居

L_NBR　是個整數變數，其值為該 CPU 左鄰的 CPU id

R_NBR　是個整數變數，其值為該 CPU 右鄰的 CPU id

CALL MPI_CART_SHIFT (COMM2D, UPDOWN, UP, B_NBR, T_NBR, IERR)

引數　UPDOWN 是個整數變數，其值設為一是要取得第二維 (I 方向) 的鄰居

UP　是個整數變數，其值設為一是要取得下、上 鄰居

B_NBR　是個整數變數，其值為該 CPU 底鄰的 CPU id

T_NBR　是個整數變數，其值為該 CPU 頂鄰的 CPU id

L_NBR、R_NBR、B_NBR、T_NBR 代表 left_neighbor、right_neighbor、botton_neighbor、top_neighbor。

5.4.3 定義固定間隔資料的 MPI 副程式

MPI_TYPE_VECTOR、MPI_TYPE_COMMIT

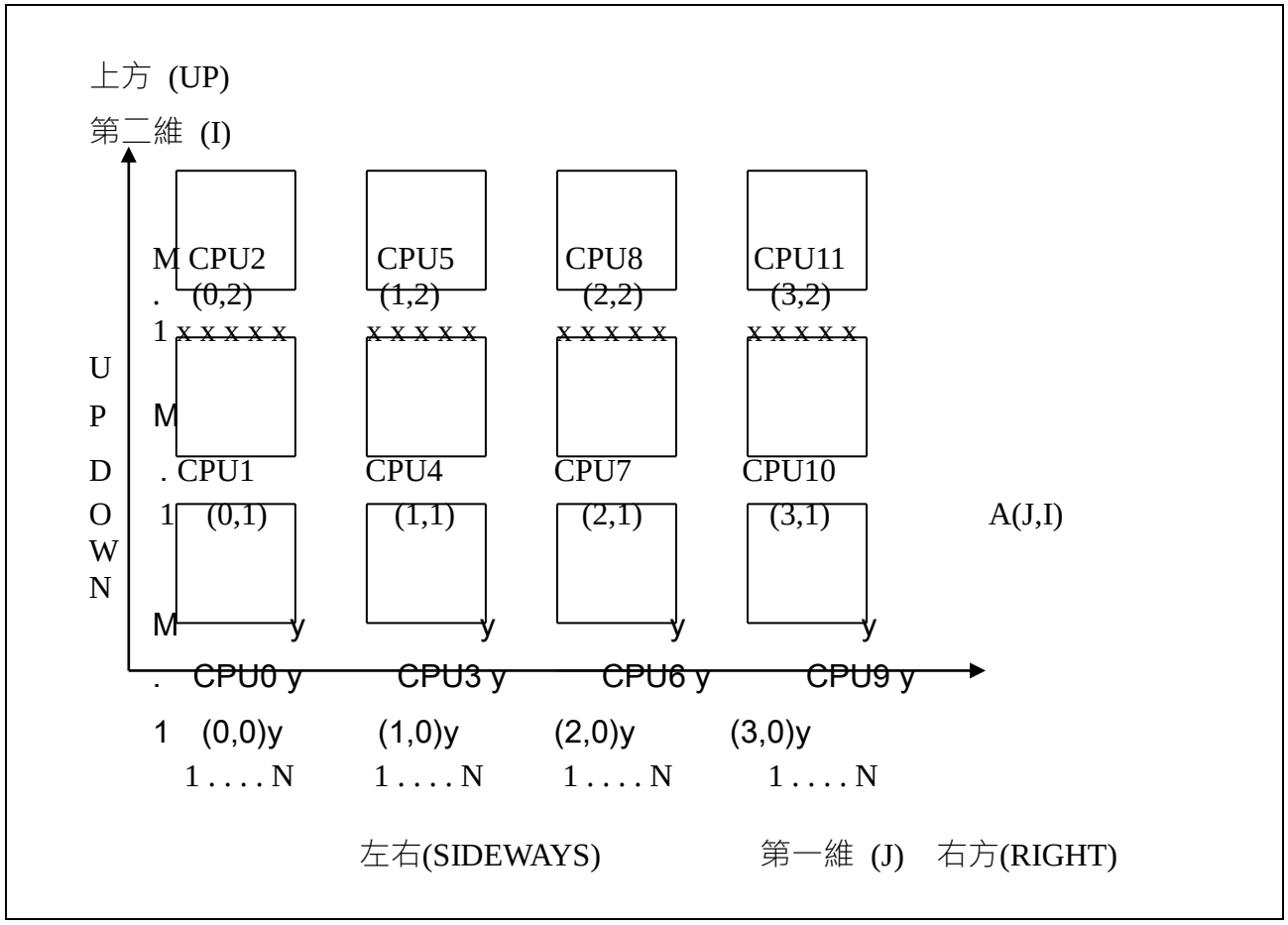


圖 5.4 二維切割邊界資料示意圖

二維陣列 A(NN,MM) 經二維切割後，有兩個方向要做資料交換。由於 Fortran 語言在記憶體上配置陣列 A 的元素時是第一維 index J 變動得最快而最後一維 I 變動得最慢的順序，如圖 5.4 所示，在各別 CPU 裏 I=1 時 J 從 1 排到 N，I=2 時 J 從 1 排到 N，直到 I=M 時 J 從 1 排到 N。第二維上下方向 I-1 或 I+1 的交換資料如圖 5.4 橫排的 x x x x 是連續位址資料。但是第一維左右方向 J-1 或 J+1 的交換資料如圖 5.4 直排的 y y y y 就不是連續位址資料，當

dimension 為 A(N,M) 時，相鄰兩個 y 相距 N 個陣列元素。

各別 CPU 要定義這樣一種固定間距的資料串 y y y y 就必須叫用 MPI_TYPE_VECTOR 副程式。每一次設定一種新的資料類別，必須再叫用一次 MPI_TYPE_COMMIT，設定的工作才告完成。MPI_TYPE_VECTOR 及 MPI_TYPE_COMMIT 的叫用格式如下 ：

```
CALL MPI_TYPE_VECTOR (COUNT, BLKLEN, STRIDE, OLDTYPE, NEWTYPE,
                      IERR)
CALL MPI_TYPE_COMMIT (NEWTYPE, IERR)
```

其引數

COUNT	是個整數變數，其值為資料串的數目
BLKLEN	是個整數變數，其值為各別資料的長度
STRIDE	是個整數變數，其值為各別資料的間距
OLDTYPE	是個整數變數，原來的資料類別
NEWTYPE	是個整數變數，資料串新的資料類別

例如要定義圖 5.4 裏的資料串 y y y y，就叫用 ：

```
CALL MPI_TYPE_VECTOR (M, 1, N, MPI_REAL, STRIDE_N, IERR)
CALL MPI_TYPE_COMMIT (STRIDE_N, IERR)
```

於是這個 y y y y 資料串的資料類別叫做 STRIDE_N。

5.5 多維陣列末二維切割的平行程式 T5_2D

循序程式 T5SEQ 的陣列資料如下 :

```
      PARAMETER     (KK=20,NN=120,MM=160, KM=3, NN1=NN-1,MM1=MM-1)

      DIMENSION      U1(KK,NN,MM ),V1(KK,NN,MM ),PS1(NN,MM)
      COMMON/BLK4/F1(KM,NN,MM),F2(KM,NN,MM),
     1                HXU(NN,MM),HXV(NN,MM),HMMX(NN,MM),HMMY(NN,MM)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)

      DIMENSION      D7(NN,MM),D8(NN,MM),D00(KK,NN,MM)
```

其中的二維陣列做二維切割的方法可參考 5.4 節 的說明。此處第一維的長度是 NN，第二維的長度是 MM，如圖 5.5 所示。三維陣列是切割其第二維和第三維，第二維的長度是 NN，第三維的長度是 MM。如果 NN 切成兩塊，MM 切成四塊，則 JP 為 2 和 IP 為 4，而且 N=NN/JP，M=MM/IP，如圖 5.5 所示，其陣列資料的宣告改為 :

```
      PARAMETER     (KK=20,NN=120,MM=160,KM=3,MM1=MM-1,NN1=NN-1)
      PARAMETER     (JP=2, IP=4, N=NN/JP, M=MM/IP, NP=JP*IP)

      DIMENSION      TT(KM,NN,MM)
      DIMENSION      U1(KK,0:N+1,0:M+1),V1(KK,0:N+1,0:M+1),PS1(0:N+1,0:M+1)
      COMMON/BLK4/F1(KM,0:N+1,0:M+1),F2(KM,0:N+1,0:M+1),
     1                HXU(0:N+1,0:M+1),HXV(0:N+1,0:M+1),
     2                HMMX(0:N+1,0:M+1),HMMY(0:N+1,0:M+1)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)

      DIMENSION      D7(0:N+1,0:M+1),D8(0:N+1,0:M+1),D00(KK,0:N+1,0:M+1)
```

其中，經過切割的 dimension 前後各加一個陣列元素位置而成為 (0:N+1, 0:M+1)。

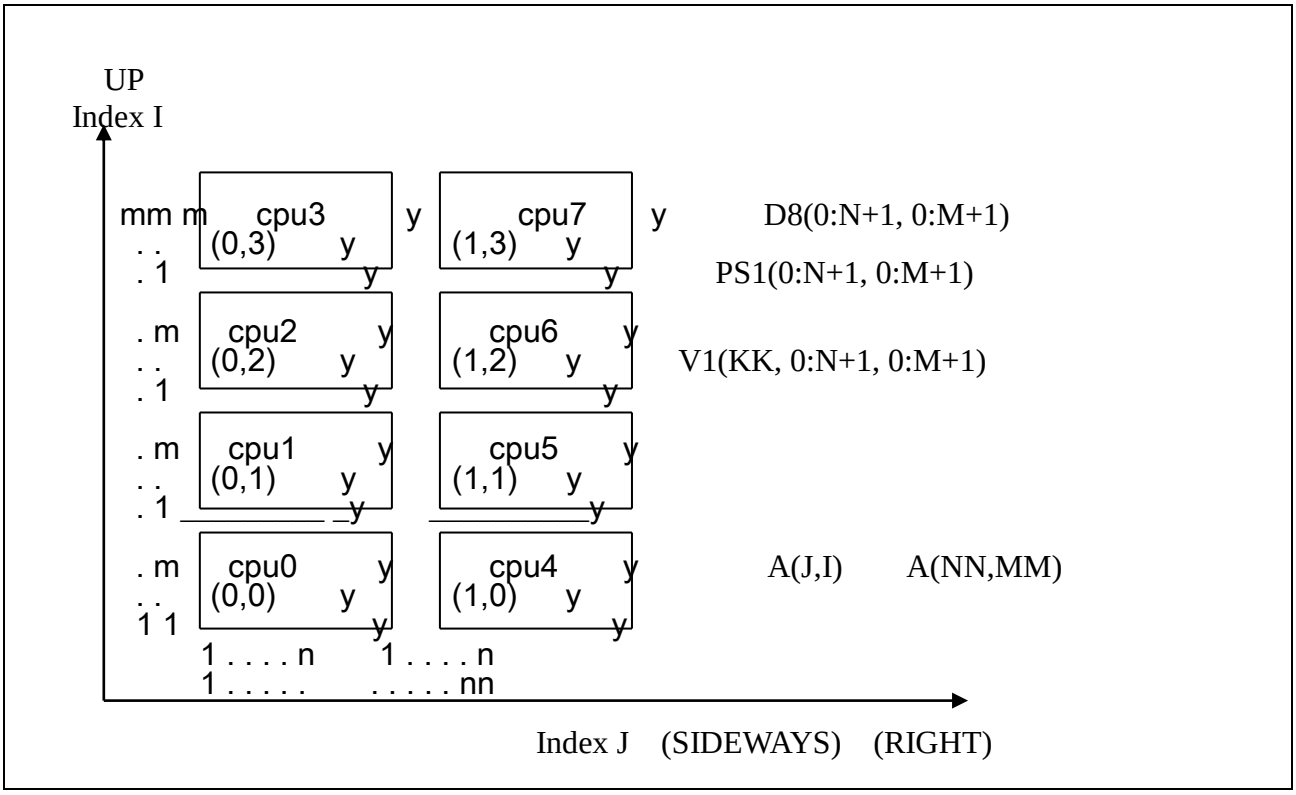


圖 5.5　多維陣列二維切割區塊圖

參考 5.4 節 的說明，可以叫用下列 MPI 副程式來設定上述的 2x4 二維切割區塊 ：

```
  SUBROUTINE NBR2D(COMM2D, MY_CID, MY_COORD, L_NBR, R_NBR,
1                                B_NBR, T_NBR, JP, IP)
  INCLUDE      'mpif.h'
  PARAMETER (NDIM=2)
  INTEGER      COMM2D, MY_CID, MY_COORD(NDIM), L_NBR, R_NBR,
1                  B_NBR, T_NBR, IPART(2), SIDEWAYS, UPDOWN, RIGHT, UP
  LOGICAL      PERIODS(2), REORDER

  IPART(1)=JP
  IPART(2)=IP
  PERIODS(1)=.FALSE.
  PERIODS(2)=.FALSE.
  REORDER=.TRUE.
  SIDEWAYS=0
  UPDOWN=1
  RIGHT=1
  UP=1

  CALL MPI_CART_CREATE (MPI_COMM_WORLD, NDIM, IPART, PERIODS,
&                                REORDER, COMM2D, IERR)
  CALL MPI_COMM_RANK (COMM2D, MY_CID, IERR)
  CALL MPI_CART_COORDS (COMM2D, MY_CID, NDIM, MY_COORD, IERR)
  CALL MPI_CART_SHIFT (COMM2D, SIDEWAYS, RIGHT, L_NBR, R_NBR, IERR)
  CALL MPI_CART_SHIFT (COMM2D, UPDOWN, UP, B_NBR, T_NBR, IERR)
  RETURN
  END
```

切割後，各個 CPU 裏要跟左右鄰居交換的資料如圖 5.5 的直排資料串 y y y y，其資料數目為 M 項。資料類別在二維陣列取名 STRIDE2D，其相鄰資料間距為 N+2，其各別資料長度為 1。在三維陣列取名 STRIDE3D，其相鄰資料間距為 KK*(N+2)，其各別資料長度為 KK。可以使用下列 MPI 指令來設定：

```
  N2=N+2
  CALL MPI_TYPE_VECTOR (M, 1, N2, MPI_REAL8, VECT_2D, IERR)
```

```
CALL MPI_TYPE_COMMIT (VECT_2D, IERR)
N2KK=N2*KK
CALL MPI_TYPE_VECTOR (M, KK, N2KK, MPI_REAL8, VECT_3D, IERR)
CALL MPI_TYPE_COMMIT (VECT_3D,IERR)
```

這樣設定之後，整串直排資料 y y y y 的資料類別在二維陣列叫做 VECT_2D，在三維陣

叫做 VECT_3D，其資料數量都是 1。J 方向 DO loop 的範圍是從 1 到 N，所以

```
JSTART=1
JEND=N
JSTARTM1=JSTART-1
JENDP1=JEND+1
```

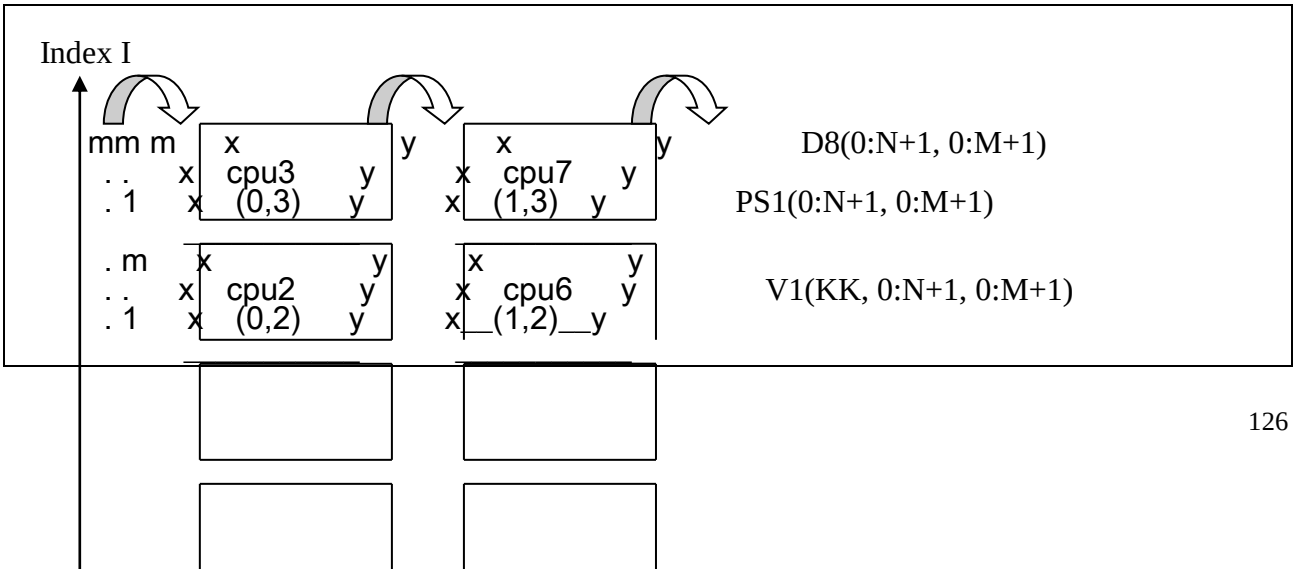
左右 CPU 之間的邊界資料交換時，如圖 5.6，當該 CPU 把最右邊的一排資料 (y 串) 送給右

鄰，同時自左鄰取得 JSTARTM1 的一排資料 (y 串)，可以使用下列 MPI_SENDRECV 來完

成 ：

```
 CALL MPI_SENDRECV (PS1(JEND,1),         1, VECT_2D, R_NBR, ITAG,
1                                PS1(JSTARTM1,1), 1, VECT_2D, L_NBR, ITAG,
2                                COMM2D, ISTATUS, IERR)

 CALL MPI_SENDRECV (V1(1,JEND,1),         1, VECT_3D, R_NBR, ITAG,
1                                V1(1,JSTARTM1,1), 1, VECT_3D, L_NBR, ITAG,
2                                COMM2D, ISTATUS, IERR)
```



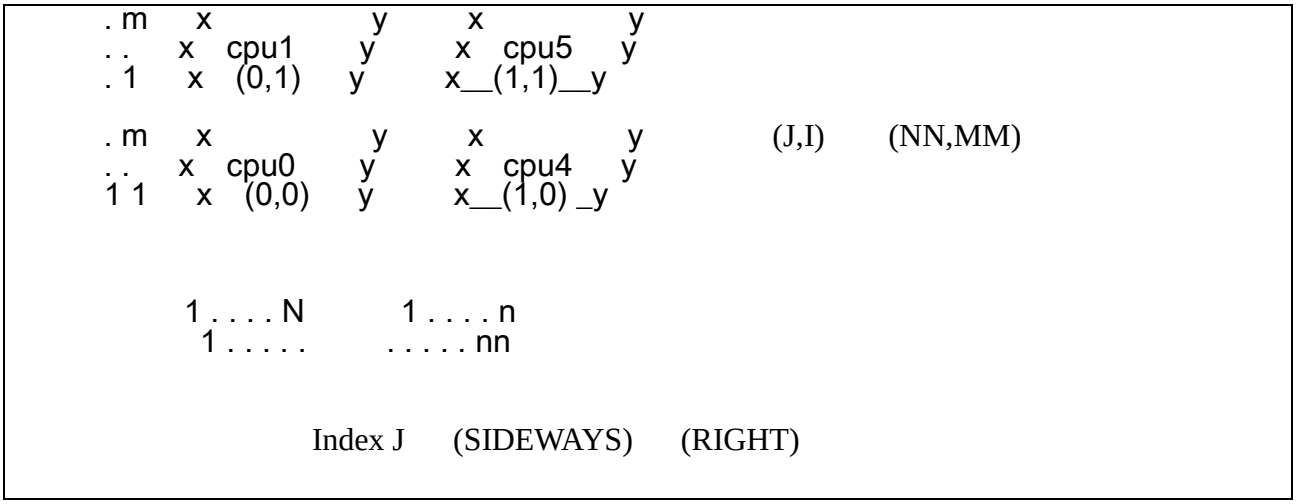


圖 5.6　多維陣列二維切割邊界資料示意圖

同理，當該 CPU 把最左邊的一排資料 (x 串) 送給左鄰，同時自右鄰取得 JENDP1 的一排資料 (x 串)，可以使用下列 MPI_SENDRECV 來完成 ：

```
 CALL MPI_SENDRECV (PS1(JSTART,1), 1, VECT_2D, L_NBR, ITAG,
1                   PS1(JENDP1,1), 1, VECT_2D, R_NBR, ITAG,
2                   COMM2D, ISTATUS, IERR)
 CALL MPI_SENDRECV (V1(1,JSTART,1), 1, VECT_3D, L_NBR, ITAG,
1                   V1(1,JENDP1,1), 1, VECT_3D, R_NBR, ITAG,
2                   COMM2D, ISTATUS, IERR)
```

至於 I 方向 DO loop 的範圍是從 1 到 M，所以

```
ISTART=1
IEND=M
ISTARTM1=ISTART-1
IENDP1=IEND+1
```

上下 CPU 之間的邊界資料交換是屬於最後一維的切割，與 5.3 節 T5DCP 程式類似，不再重復。例如，當該 CPU 把最底邊的一排資料送給底鄰，同時自頂鄰取得 IENDP1 的一排資料，

可以使用下列 MPI_SENDRECV 來完成 ：

```
      CALL MPI_SENDRECV (PS1(1,ISTART), N, MPI_REAL8, B_NBR, ITAG,
     1                   PS1(1,IENDP1), N, MPI_REAL8, T_NBR, ITAG,
     4                   COMM2D, ISTATUS, IERR)
```

當該 CPU 把最上邊的一面資料送給頂鄰，同時自底鄰取得 ISTARTM1 的一面資料，可以使用下列 MPI_SENDRECV 來完成 ：

```
      N2KK=(N+2)*KK
      CALL MPI_SENDRECV (U1(1,1,IEND ),      N2KK, MPI_REAL8, T_NBR, ITAG,
     1                   U1(1,1,ISTARTM1), N2KK, MPI_REAL8, B_NBR, ITAG,
     2                   COMM2D, ISTATUS, IERR)
```

陣列 TT 的 dimension 與陣列 F1、F2 的 dimension 不相配，不能夠使用 MPI_GATHER 或 MPI_GATHERV 來收集計算的結果。只能夠使用 MPI_SEND、MPI_RECV 來傳送各個 CPU 計算出來的結果，再用一個 COPY1 副程式來把 F2 的資料移入 TT。

```
      DIMENSION     TT(KM,NN,MM)
      COMMON/BLK4/F1(KM,0:N+1,0:M+1),F2(KM,0:N+1,0:M+1),
```

多維陣列末二維切割的平行程式 T5_2D 如下 ：

```
      PROGRAM    T5_2D
      IMPLICIT REAL*8 (A-H,O-Z)
      INCLUDE 'mpif.h'
C   - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
C        2-D data partition on the last two dimension of
C        multi-dimensional array with -1,+1 data exchange
C   - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
```

```
      PARAMETER     (KK=20,NN=120,MM=160, KM=3,NN1=NN-1,MM1=MM-1)
      PARAMETER     (JP=2, IP=4, N=NN/JP, M=MM/IP, NP=JP*IP)
      DIMENSION      TT(KM,NN,MM)
      DIMENSION      U1(KK,0:N+1,0:M+1),V1(KK,0:N+1,0:M+1),PS1(0:N+1,0:M+1)
      COMMON/BLK4/F1(KM,0:N+1,0:M+1),F2(KM,0:N+1,0:M+1),
     1                HXU(0:N+1,0:M+1),HXV(0:N+1,0:M+1),
     2                HMMX(0:N+1,0:M+1),HMMY(0:N+1,0:M+1)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)
      DIMENSION      D7(0:N+1,0:M+1),D8(0:N+1,0:M+1),D00(KK,0:N+1,0:M+1)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       mpi related data, this code is designed for up to NP processors
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      PARAMETER (NDIM=2)
      INTEGER     ISTATUS(MPI_STATUS_SIZE), COMM2D,
     1            L_NBR, R_NBR, B_NBR, T_NBR, MY_CID, MY_COORD(NDIM),
     2            VECT_2D, VECT_3D
      COMMON/BLKMPI/MYID, NPROC, ISTATUS, L_NBR, R_NBR, B_NBR, T_NBR,
     1            MY_CID, STRIDE2D, STRIDE3D,
     2            ISTART, ISTART2, IEND, IEND1, ISTARTM1, IENDP1,
     3            JSTART, JSTART2, JEND, JEND1, JSTARTM1, JENDP1,
     4            ISTARTG(0:NP),IENDG(0:NP),JSTARTG(0:NP),JENDG(0:NP)
C       CHARACTER*30    NODENAME
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       mpi related data will be set by main program, and pass them to
C       all subprograms by common block /BLKMPI/
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      IF(NPROC.NE.NP) THEN
        PRINT *,' NPROC NOT EQUAL TO ', NP,' PROGRAM WILL STOP'
        STOP
      ENDIF
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL NBR2D(COMM2D,MY_CID,MY_COORD,L_NBR,R_NBR,B_NBR,T_NBR,
     1            JP,IP)
      CALL MPI_BARRIER (COMM2D,IERR)
      CLOCK=MPI_WTIME()
      CALL STARTEND (JP,1,NN,JSTARTG, JENDG, JCOUNTG)
```

```
      CALL STARTEND (IP,1,MM,ISTARTG, IENDG, ICOUNTG)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      ISTART=1
      IEND=M
      JSTART=1
      JEND=N
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C         for DO I=x,MM1 (MM-1)
C         for DO J=x,NN1 (NN-1)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  ---
      IEND1=IEND
      JEND1=JEND
      IF( MY_COORD(2).EQ.IP-1) IEND1=IEND1-1
      IF( MY_COORD(1).EQ.JP-1) JEND1=JEND1-1
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C         for DO I=2,x
C         for DO J=2,x
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      ISTART2=ISTART
      JSTART2=JSTART
      IF( MY_COORD(2).EQ.0) ISTART2=2
      IF( MY_COORD(1).EQ.0) JSTART2=2
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C         for ghost point
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      ISTARTM1=ISTART-1
      IENDP1=IEND+1
      JSTARTM1=JSTART-1
      JENDP1=JEND+1
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Test data generation
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
CCC   DO 10 I=1,MM1
      DO 10 I=1,IEND1
      II=I+ISTARTG(MY_COORD(2))-1
CCC   DO 10 J=1,NN
      DO 10 J=1,JEND
      JJ=J+JSTARTG(MY_COORD(1))-1
```

```
      DO 10 K=1,KK
        U1(K,J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(JJ)+1.D0/DFLOAT(K)
   10 CONTINUE
CCC   DO 20 I=1,MM
      DO 20 I=1,IEND
      II=I+ISTARTG(MY_COORD(2))-1
CCC   DO 20 J=1,NN1
      DO 20 J=1,JEND1
      JJ=J+JSTARTG(MY_COORD(1))-1
      DO 20 K=1,KK
        V1(K,J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(JJ)+1.D0/DFLOAT(K)
   20 CONTINUE
CCC   DO 30 I=1,MM
      DO 30 I=1,IEND
      II=I+ISTARTG(MY_COORD(2))-1
CCC   DO 30 J=1,NN
      DO 30 J=1,JEND
        JJ=J+JSTARTG(MY_COORD(1))-1
        PS1(J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(JJ)
        HXU(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(JJ)
        HXV(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(JJ)
        HMMX(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(JJ)
        HMMY(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(JJ)
   30 CONTINUE

      DO 40 K=1,KK
      AM7(K)=1.D0/DFLOAT(K)
      DO 40 KA=1,KK
        VECINV(KA,K)=1.D0/DFLOAT(KA)+1.D0/DFLOAT(K)
   40 CONTINUE
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Start the computation
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      N2=N+2
      N2KK=N2*KK
      CALL MPI_TYPE_VECTOR (M, KK, N2KK, MPI_REAL8, IVECT_3D, IERR)
      CALL MPI_TYPE_COMMIT (IVECT_3D, MPI_ERR)
      CALL MPI_TYPE_VECTOR (M, 1, N2, MPI_REAL8, IVECT_2D, IERR)
```

```
      CALL MPI_TYPE_COMMIT (IVECT_2D, IERR)
      CALL MPI_BARRIER (COMM2D, IERR)
      ITAG=10
      CALL MPI_SENDRECV (U1(1,1,IEND ),       N2KK, MPI_REAL8, T_NBR, ITAG,
     1                   U1(1,1,ISTARTM1), N2KK, MPI_REAL8, B_NBR, ITAG,
     2                   COMM2D, ISTATUS, IERR)
      ITAG=20
      CALL MPI_SENDRECV(V1(1,JEND,1 ),       1, IVECT_3D, R_NBR, ITAG,
     1                   V1(1,JSTARTM1,1), 1, IVECT_3D, L_NBR, ITAG,
     2                   COMM2D, ISTATUS, IERR)
      ITAG=30
      CALL MPI_SENDRECV (PS1(1,ISTART), N, MPI_REAL8, B_NBR, ITAG,
     1                    PS1(1,IENDP1), N, MPI_REAL8, T_NBR, ITAG,
     2                    COMM2D, ISTATUS, IERR)
      ITAG=40
      CALL MPI_SENDRECV (PS1(JSTART,1), 1, IVECT_2D, L_NBR, ITAG,
     1                    PS1(JENDP1,1), 1, IVECT_2D, R_NBR, ITAG,
     2                    COMM2D, ISTATUS, IERR)
CCC   DO 210 I=1,MM
CCC   DO 210 J=1,NN
      DO 210 I=ISTART,IEND
      DO 210 J=JSTART,JEND
      DO 210 K=1,KM
        F1(K,J,I)=0.0D0
        F2(K,J,I)=0.0D0
 210  CONTINUE


CCC   DO 220 I=1,MM1
CCC   DO 220 J=2,NN1
      DO 220 I=ISTART,IEND1
      DO 220 J=JSTART2,JEND1
        D7(J,I)=(PS1(J,I+1)+PS1(J,I))*0.5D0*HXU(J,I)
 220  CONTINUE

CCC   DO 230 I=2,MM1
CCC   DO 230 J=1,NN1
      DO 230 I=ISTART2,IEND1
```

```
      DO 230 J=JSTART,JEND1
        D8(J,I)=(PS1(J+1,I)+PS1(J,I))*0.5D0*HXV(J,I)
  230 CONTINUE
      CALL MPI_BARRIER(COMM2D, MPIERROR)
      ITAG=50
      CALL MPI_SENDRECV (D7(1,IEND),        N, MPI_REAL8, T_NBR, ITAG,
     1                   D7(1,ISTARTM1), N, MPI_REAL8, B_NBR, ITAG,
     2                   COMM2D, ISTATUS, IERR)
      ITAG=60
      CALL MPI_SENDRECV (D8(JEND,1),        1, IVECT_2D, R_NBR, ITAG,
     1                   D8(JSTARTM1,1), 1, IVECT_2D, L_NBR, ITAG,
     2                   COMM2D, ISTATUS, IERR)

CCC   DO 240 I=2,MM1
CCC   DO 240 J=2,NN1
      DO 240 I=ISTART2,IEND1
      DO 240 J=JSTART2,JEND1
      DO 240 K=1,KK
        D00(K,J,I)=(D7(J,I)*U1(K,J,I)-D7(J,I-1)*U1(K,J,I-1))*HMMX(J,I)
     1            +(D8(J,I)*V1(K,J,I)-D8(J-1,I)*V1(K,J-1,I))*HMMY(J,I)
  240 CONTINUE


CCC   DO 260 I=2,MM1
      DO 260 I=ISTART2,IEND1
      DO 260 KA=1,KK
CCC   DO 260 J=2,NN1
      DO 260 J=JSTART2,JEND1
      DO 260 K=1,KM
        F1(K,J,I)=F1(K,J,I)-VECINV(K,KA)*D00(KA,J,I)
  260 CONTINUE
      SUMF1=0.D0
      SUMF2=0.D0
CCC   DO 270 I=2,MM1
      DO 270 I=ISTART2,IEND1
CCC   DO 270 J=2,NN1
      DO 270 J=JSTART2,JEND1
      DO 270 K=1,KM
```

```
            F2(K,J,I)=-AM7(K)*PS1(J,I)
            SUMF1=SUMF1+F1(K,J,I)
            SUMF2=SUMF2+F2(K,J,I)
  270    CONTINUE
C   - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
C         Output data for validation
C   - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
          CALL MPI_BARRIER (COMM2D,IERR)
          IROOT=0
          CALL MPI_REDUCE (SUMF1, GSUMF1, 1, MPI_REAL8, MPI_SUM,
         1                           IROOT, COMM2D, IERR)
          CALL MPI_REDUCE (SUMF2, GSUMF2, 1, MPI_REAL8, MPI_SUM,
         1                           IROOT, COMM2D, IERR)
          KOUNT=KM*(N+2)*(M+2)
          ITAG=70
          IF (MY_CID.NE.0) THEN
            CALL MPI_SEND (F2, KOUNT, MPI_REAL8, IROOT, ITAG, COMM2D, IERR)
          ELSE
            CALL COPY1(MY_CID, F2, TT,ISTARTG,JSTARTG)
            DO ISRC=1,NPROC-1
              CALL MPI_RECV (F2, KOUNT, MPI_REAL8, ISRC, ITAG,
         1                            COMM2D, ISTATUS, IERR)
              CALL COPY1 (ISRC, F2, TT,ISTARTG,JSTARTG)
            ENDDO
          ENDIF
  301    FORMAT(8E10.3)
          IF(MY_CID.EQ.0) THEN
            PRINT *,'SUMF1,SUMF2=', GSUMF1,GSUMF2
            PRINT *,' F2(2,2,I),I=1, 160,5'
            PRINT 301,(TT(2,2,I),I=1, 160,5)
          ENDIF
          CLOCK=MPI_WTIME() - CLOCK
          PRINT *,' MY_CID, CLOCK TIME=', MY_CID,CLOCK
          CALL MPI_FINALIZE (IERR)
          STOP
          END
          SUBROUTINE NBR2D(COMM2D, MY_CID, MY_COORD, L_NBR, R_NBR,
         &                           B_NBR, T_NBR, JP, IP)
```

```
      INCLUDE     'mpif.h'
      PARAMETER (NDIM=2)
      INTEGER     COMM2D, MY_CID, MY_COORD(NDIM), L_NBR, R_NBR,
     &            B_NBR, T_NBR, JP, IP
      INTEGER     IPART(2), SIDEWAYS, UPDOWN, RIGHT, UP
      LOGICAL     PERIODS(2), REORDER
C
      IPART(1)=JP
      IPART(2)=IP
      PERIODS(1)=.FALSE.
      PERIODS(2)=.FALSE.
      REORDER=.TRUE.
      SIDEWAYS=0
      UPDOWN=1
      RIGHT=1
      UP=1
      CALL MPI_CART_CREATE ( MPI_COMM_WORLD, NDIM,I PART, PERIODS,
     1                       REORDER, COMM2d, MPI_ERR)
      CALL MPI_COMM_RANK ( COMM2D, MY_CID, MPI_ERR)
      CALL MPI_CART_COORDS( COMM2D, MY_CID, NDIM, MY_COORD, MPI_ERR)
      CALL MPI_CART_SHIFT( COMM2D,SIDEWAYS,RIGHT,L_NBR,R_NBR,MPI_ERR)
      CALL MPI_CART_SHIFT( COMM2D, UPDOWN, UP, B_NBR, T_NBR, MPI_ERR)
      PRINT *,'MY_CID=',MY_CID,', COORD=(',MY_COORD(1),',',MY_COORD(2),')',
     1     ', L_,R_,T_,B_NBR=', L_NBR,R_NBR,T_NBR,B_NBR
      RETURN
      END
      SUBROUTINE COPY1(ISRC,FF,TT,ISTARTG,JSTARTG)
      INCLUDE    'mpif.h'
C  - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
C       Copy partitioned array FF to global array TT
C  - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
      PARAMETER    (KK=20,NN=120,MM=160, KM=3,MM1=MM-1,NN1=NN-1)
      PARAMETER    (JP=2, IP=4, N=NN/JP, M=MM/IP, NP=IP*JP)
      REAL*8        FF(KM,0:N+1,0:M+1),TT(KM,NN,MM),
     1              ISTARTG(0:NP),JSTARTG(0:NP)
      IF(ISRC.LT.IP) THEN
         JJ=0
         II=ISRC
```

```
      ELSE
         JJ=ISRC/IP
         II=ISRC-JJ*IP
      ENDIF
      DO I=1,M
         IG=ISTARTG(II)+I-1
         DO J=1,N
            JG=JSTARTG(JJ)+J-1
            DO K=1,KM
               TT(K,JG,IG)=FF(K,J,I)
            ENDDO
         ENDDO
      ENDDO
      RETURN
      END
```

平行程式 T5_2D 的測試結果如下 ：

```
 ATTENTION: 0031-408    8 nodes allocated by LoadLeveler, continuing...
   MY_CID= 0 , COORD=( 0 , 0 ), L_,R_,T_,B_NBR= -3    4    1    -3
   MY_CID= 1 , COORD=( 0 , 1 ), L_,R_,T_,B_NBR= -3    5    2    0
   MY_CID= 2 , COORD=( 0 , 2 ), L_,R_,T_,B_NBR= -3    6    3    1
   MY_CID= 4 , COORD=( 1 , 0 ), L_,R_,T_,B_NBR= 0    -3    5    -3
   MY_CID= 5 , COORD=( 1 , 1 ), L_,R_,T_,B_NBR= 1    -3    6    4
   MY_CID= 7 , COORD=( 1 , 3 ), L_,R_,T_,B_NBR= 3    -3    -3    6
   MY_CID= 3 , COORD=( 0 , 3 ), L_,R_,T_,B_NBR= -3    7    -3    2
   MY_CID= 6 , COORD=( 1 , 2 ), L_,R_,T_,B_NBR= 2    -3    7    5
     MY_CID, CLOCK TIME= 1    0.8523987489752471 45E-01
     MY_CID, CLOCK TIME= 3    0.9445869992487132 55E-01
     MY_CID, CLOCK TIME= 2    0.8980842493474483 49E-01
     MY_CID, CLOCK TIME= 5    0.1043841249775141 48
     MY_CID, CLOCK TIME= 4    0.1003765249624848 37
     MY_CID, CLOCK TIME= 7    0.1205861000344157 22
     MY_CID, CLOCK TIME= 6    0.1168324500322341 92
 SUMF1,SUMF2= 26172.4605364287345    -2268.89180334310413
     F2(2,2,I),I=1,160,5
     .000E+00 -.333E+00 -.295E+00 -.281E+00 -.274E+00 -.269E+00 -.266E+00 -.264E+00
```

```
 -.262E+00 -.261E+00 -.260E+00 -.259E+00 -.258E+00 -.258E+00 -.257E+00 -.257E+00
 -.256E+00 -.256E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.254E+00
 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.253E+00 -.253E+00 -.253E+00
   MY_CID, CLOCK TIME= 0    0.125097200041636825
```

執行循序程式 t5seq 需時 0.36 秒，在八個 CPU 上執行平行程式 t5_2d 需時 0.125 秒，平行效率 (parallel speed up) 為 0.36/0.125 = 2.88 倍。在八個 CPU 上平行計算時，採用末維資料切割方法需時 0.082 秒，而採用二維資料切割方法需時 0.125 秒。由此可知，二維資料切割的平行效率比末維資料切割的平行效率稍差。其原因有二，其一是末維資料切割時只需做左右方向的邊界資料交換，而二維資料切割時除了左右方向的邊界資料交換之外，還要做上下方向的邊界資料交換。其二是末維資料切割時的邊界資料交換為連續位址資料交換，而二維資料切割時除了連續位址資料交換之外，還有不連續位址邊界資料的交換。所以，採用二維資料切割方法比採用末維資料切割方法在邊界資料交換方面費時較多些。

第六章　MPI 程式的效率提昇

第一節介紹 Nonblocking 資料傳送，並且與 blocking 資料傳送的效果做比較。

第二節介紹傳送資料的合併，以減少資料傳送的次數，來提高資料傳送的效率。

第三節介紹以較高速的邊界資料計算取代較低速的邊界資料交換，來提高平行計算的效率。

第四節介紹事先切割輸入資料與事後收集切割過的輸出資料。

6.1 Nonblocking 資料傳送

在前面幾章介紹的 MPI_SEND、MPI_RECV 都是屬於 Blocking 型式的點對點通訊。在執行 MPI_SEND 指令時，送出資料的 CPU 要等到該資料完全送出去（Buffer is empty）之後 MPI_SEND 才算完成，然後才執行其後的指令。同理，在執行 MPI_RECV 指令時，接收資料的 CPU 要等到該資料完全收齊（Buffer is full）之後 MPI_RECV 才算完成。請參閱圖 6.1 的 Blocking SEND/RECV 示意圖。

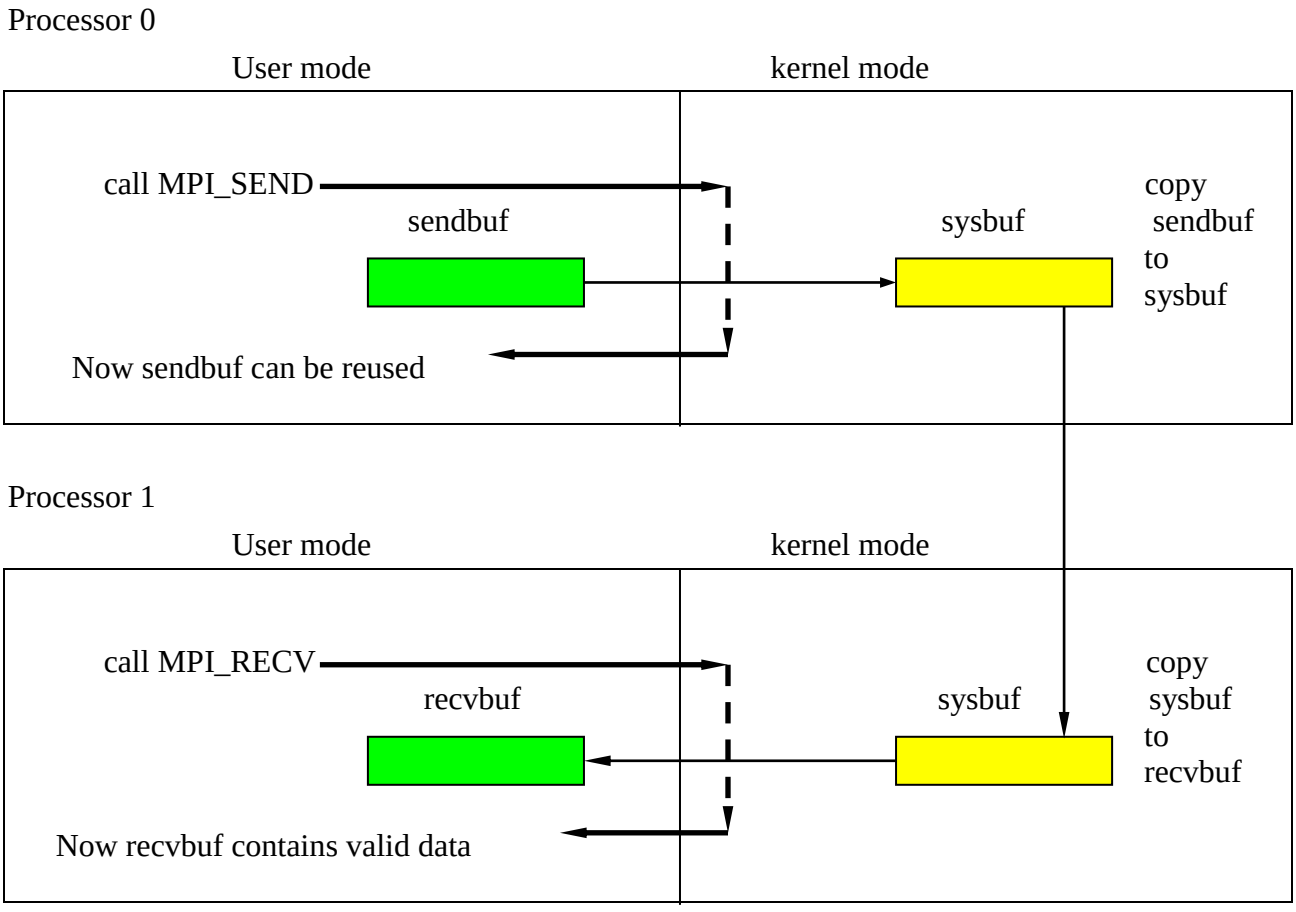


圖 6.1 Blocking SEND/RECV 示意圖

另外一種型式稱為 Nonblocking 點對點通訊，計有 MPI_ISEND、MPI_IRECV 兩個指令，在執行這種指令時不需要等該資料送畢或收齊，立刻接著執行其後的指令。等到遇到 MPI_WAIT 指令時，才停下來查驗該送出或接收工作是否已經完畢，等到該資料完全送畢或收齊之後，才執行其後的指令。於是在 MPI_ISEND 或 MPI_IRECV 指令之後與 MPI_WAIT 指令之前可以執行其他計算指令，在這一段時間裏資料傳送與計算工作可以並行不悖。如果這一段時間夠長，資料傳送與計算工作並行的效果就會顯現出來。請參閱圖 6.2 的 Nonblocking SEND/RECV 示意圖。

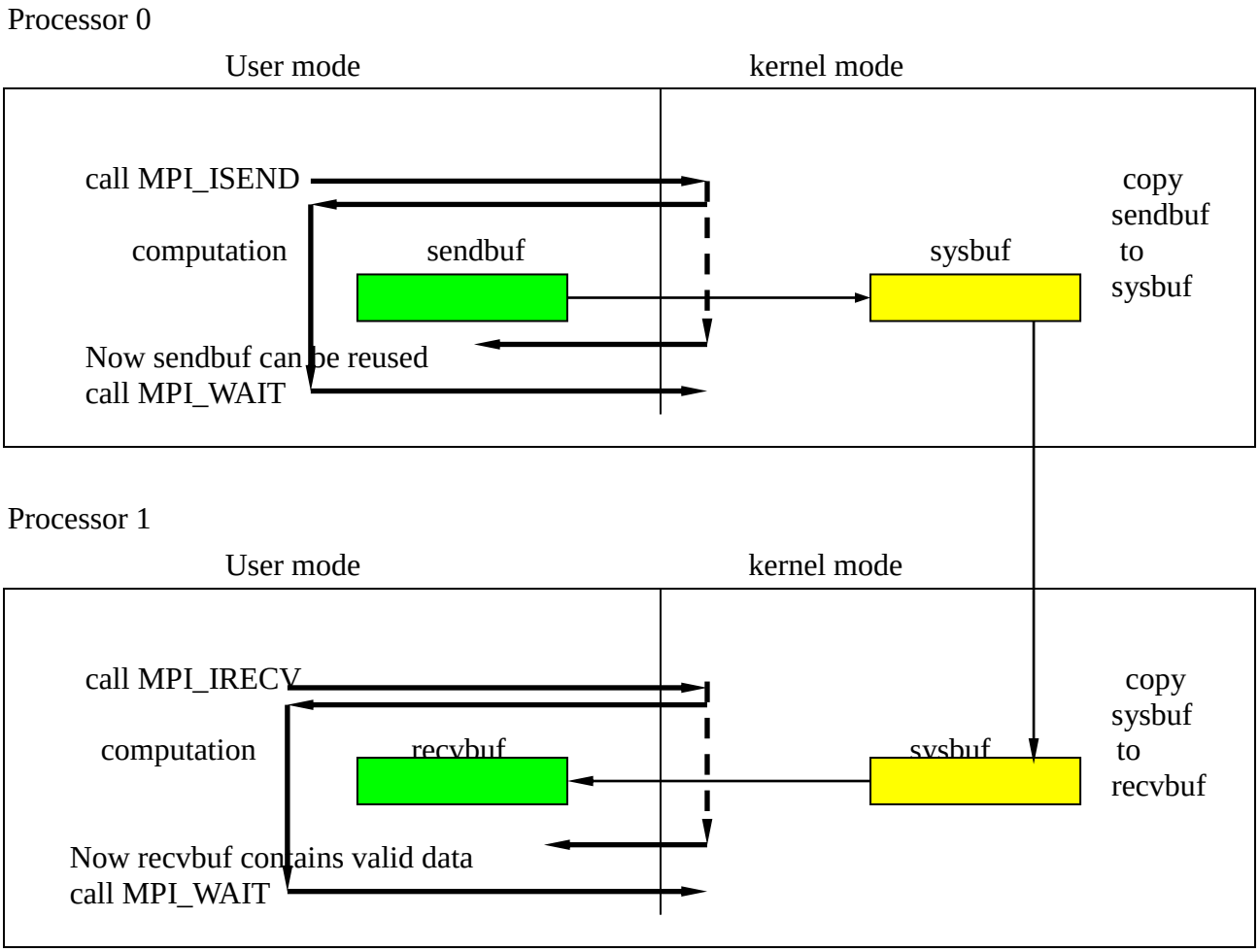


圖 6.2 Nonblocking SEND/RECV 示意圖

MPI_ISEND 的叫用格式如下

```
      CALL MPI_ISEND (DATA, COUNT, DATATYPE, DEST, TAG,
     &                MPI_COMM_WORLD, REQUEST, IERR)
```

其引數

DATA	是要送出的資料起點
COUNT	是要送出的資料數量
DATATYPE	是要送出的資料類別
DEST	是收受資料的 CPU id
TAG	是要送出資料的標籤
REQUEST	是此次資料傳送的編號
MPI_COMM_WORLD	是內定的 communicator

MPI_IRECV 的叫用格式如下

```
      CALL MPI_IRECV (DATA, COUNT, DATATYPE, SRC, TAG,
     1                MPI_COMM_WORLD, REQUEST, IERR)
```

其引數

DATA	是收受的資料起點

COUNT 是收受的資料數量

DATATYPE 是收受的資料類別

DEST 是送出資料的 CPU id

TAG 是收受資料的標籤

MPI_COMM_WORLD 是內定的 communicator

REQUEST 是此次資料傳送的編號

MPI_WAIT 的叫用格式如下

CALL MPI_WAIT (REQUEST, ISTATUS, IERR)

其引數

REQUEST 是要查詢的資料傳送編號，必須與對應的 MPI_ISEND 或 MPI_IRECV 所使用的 REQUEST 相同。

ISTATUS 是此指令執行的結果

例如 5.3 節的 T5DCP 程式裏最前面的兩個 MPI_SENDRECV 可以分別改用一個 MPI_ISEND 和一個 MPI_IRECV 取代它來傳送 PS1 和 U1 陣列的邊界資料。然後在要用到 PS1 的邊界資料的 loop-220 之前使用 MPI_WAIT 指令等待 PS1 邊界資料傳送完畢之後，才執行其計算工作。而 loop-230 不需要用到 PS1 的邊界資料，就把它移到 loop-220 之前，以增加與資料傳送重疊的計算量。在 loop-240 要用到 U1 的邊界資料，因此在 loop-220 之後使用 MPI_WAIT 指令等待 U1 邊界資料傳送完畢。在 loop-220 裏產生的陣列 D7 在 loop-240 就馬上要用到其邊

界資料，所以這一個 MPI_SENDRECV 就沒有必要改用 Nonblocking 型式的資料傳送指令了。

程式 T5DCP 經過這樣改寫而成為 T6DCP 如下

```
      PROGRAM    T6DCP
      IMPLICIT REAL*8 (A-H,O-Z)

      INCLUDE    'mpif.h'
C-----------------------------------------------------------------------
C       Data partition on the last dimension of muti-dimensional arrays
C       with -1,+1 data exchange and nonblocking send/receive
C       NP=8 MUST be modified when parallelized on other than 8 processors
C
C       I : U1,V1,PS1,HXU,HXV,HMMX,HMMY,VECINV,AM7
C       O : F1,F2
C       W : D7,D8,D00
C-----------------------------------------------------------------------
      PARAMETER    (KK=20,NN=120,MM=160, KM=3,MM1=MM-1,NN1=NN-1)
      PARAMETER    (NP=8, M=MM/NP+1)
      DIMENSION       TT(KM,NN,MM)
      DIMENSION       U1(KK,NN,0:M+1),V1(KK,NN,M),PS1(NN,0:M+1)
      COMMON/BLK4/F1(KM,NN,M),F2(KM,NN,M),
     1                    HXU(NN,M),HXV(NN,M),
     2                    HMMX(NN,M),HMMY(NN,M)
      COMMON/BLK5/VECINV(KK,KK),AM7(KK)

      DIMENSION       D7(NN,0:M+1),D8(NN,M),D00(KK,NN,M)
C  - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
C       mpi related data, this code is designed for up to NP processors
C  - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
      INTEGER             L_NBR, R_NBR, ISTATUS(MPI_STATUS_SIZE),
     &                    GSTART, GCOUNT, GDISP, GEND, MYCOUNT
      COMMON/BLKMPI/MYID, NPROC, ISTATUS, L_NBR, R_NBR,
```

```
     1                    ISTART, ISTART2, ISTART3, IEND, IEND1,
     2                    ISTARTM1, IENDM1, IENDP1, MYCOUNT ,
     3                    GSTART(0:NP), GEND(0:NP), GCOUNT(0:NP), GDISP(0:NP)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       mpi related data will be set by main program, and pass them to
C       all subprograms by common block /BLKMPI/
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
       CALL MPI_INIT (IERR)
       CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
       CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
       CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)
       CLOCK=MPI_WTIME()
       CALL STARTEND(NPROC, 1, MM, GSTART, GEND, GCOUNT)
       DO I=0,NPROC-1
          GCOUNT(I)=KM*NN*GCOUNT(I)
          GDISP(I)=KM*NN*(GSTART(I)-1)
       ENDDO
       ISTARTG=GSTART(MYID)
       IENDG=GEND(MYID)
       MYCOUNT=IENDG-ISTARTG+1
       PRINT *,' MYID, NPROC, ISTARTG, IENDG=', MYID, NPROC, ISTARTG, IENDG
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C**** Assume that each processor hold at least 3 grid points
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
       ISTART=1
       IEND=MYCOUNT
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C          for DO I=x,MM1 (MM-1)
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
       IEND1=IEND
       IF( MYID.EQ.NPROC-1) IEND1=IEND1-1
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C          for DO I=2,x
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
       ISTART2=1
       IF( MYID.EQ.0) ISTART2=2
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
       ISTARTM1=ISTART-1
```

```
      IENDP1=IEND+1
      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)              L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.NPROC-1)    R_NBR=MPI_PROC_NULL
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Test data generation
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
CCC   DO 10 I=1,MM1
      DO 10 I=1,IEND1
      II=I+ISTARTG-1
      DO 10 J=1,NN
      DO 10 K=1,KK
        U1(K,J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   10 CONTINUE
CCC   DO 20 I=1,MM
      DO 20 I=1,IEND
      II=I+ISTARTG-1
      DO 20 J=1,NN1
      DO 20 K=1,KK
        V1(K,J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)+1.D0/DFLOAT(K)
   20 CONTINUE
CCC   DO 30 I=1,MM
      DO 30 I=1,IEND
      II=I+ISTARTG-1
      DO 30 J=1,NN
        PS1(J,I)=1.D0/DFLOAT(II)+1.D0/DFLOAT(J)
        HXU(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)
        HXV(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(J)
        HMMX(J,I)=2.D0/DFLOAT(II)+1.D0/DFLOAT(J)
        HMMY(J,I)=1.D0/DFLOAT(II)+2.D0/DFLOAT(J)
   30 CONTINUE
      DO 40 K=1,KK
      AM7(K)=1.D0/DFLOAT(K)
      DO 40 KA=1,KK
        VECINV(KA,K)=1.D0/DFLOAT(KA)+1.D0/DFLOAT(K)
   40 CONTINUE
C  - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

```
C        Start the computation
C  - -   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
         CALL MPI_BARRIER(MPI_COMM_WORLD,MPI_ERR)
         CLOCK=MPI_WTIME()
         ITAG=10
C        CALL MPI_SENDRECV (PS1(1,ISTART), NN, MPI_REAL8, L_NBR, ITAG,
C       1                   PS1(1,IENDP1), NN, MPI_REAL8, R_NBR, ITAG,
C       2                   MPI_COMM_WORLD, ISTATUS, IERR)
         CALL MPI_ISEND (PS1(1,ISTART), NN, MPI_REAL8, L_NBR, ITAG,
        1                MPI_COMM_WORLD, IRQPS1, IERR)
         CALL MPI_IRECV (PS1(1,IENDP1), NN, MPI_REAL8, R_NBR, ITAG,
        1                MPI_COMM_WORLD, IRQPS1, IERR)
         ITAG=30
C        CALL MPI_SENDRECV(U1(1,1,IEND ),     NNKK,MPI_REAL8,R_NBR,ITAG,
C       1                   U1(1,1,ISTARTM1),NNKK,MPI_REAL8,L_NBR,ITAG,
C       2                   MPI_COMM_WORLD, ISTATUS, IERR)
         NNKK=NN*KK
         CALL MPI_ISEND (U1(1,1,IEND), NNKK, MPI_REAL8, R_NBR, ITAG,
        1                MPI_COMM_WORLD, IRQU1, IERR)
         CALL MPI_IRECV (U1(1,1,ISTARTM1), NNKK, MPI_REAL8, L_NBR, ITAG,
        1                MPI_COMM_WORLD, IRQU1, IERR)
CCC      DO 210 I=1,MM
         DO 210 I=ISTART,IEND
         DO 210 J=1,NN
         DO 210 K=1,KM
            F1(K,J,I)=0.0D0
            F2(K,J,I)=0.0D0
  210    CONTINUE
CCC      DO 230 I=2,MM1
         DO 230 I=ISTART2,IEND1
         DO 230 J=1,NN1
            D8(J,I)=(PS1(J+1,I)+PS1(J,I))*0.5D0*HXV(J,I)
  230    CONTINUE
         CALL MPI_WAIT (IRQPS1, ISTATUS, IERR)
CCC      DO 220 I=1,MM1
         DO 220 I=ISTART,IEND1
         DO 220 J=2,NN1
            D7(J,I)=(PS1(J,I+1)+PS1(J,I))*0.5D0*HXU(J,I)
```

```
 220  CONTINUE
      CALL MPI_WAIT ( IRQU1, ISTATUS, IERR)
      ITAG=50
      CALL MPI_SENDRECV (D7(1,IEND),      NN, MPI_REAL8, R_NBR, ITAG,
     1                   D7(1,ISTARTM1), NN, MPI_REAL8, L_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
CCC   DO 240 I=2,MM1
      DO 240 I=ISTART2,IEND1
      DO 240 J=2,NN1
      DO 240 K=1,KK
        D00(K,J,I)=(D7(J,I)*U1(K,J,I)-D7(J,I-1)*U1(K,J,I-1))*HMMX(J,I)
     1             +(D8(J,I)*V1(K,J,I)-D8(J-1,I)*V1(K,J-1,I))*HMMY(J,I)
 240  CONTINUE
CCC   DO 260 I=2,MM1
      DO 260 I=ISTART2,IEND1
      DO 260 KA=1,KK
      DO 260 J=2,NN1
      DO 260 K=1,KM
        F1(K,J,I)=F1(K,J,I)-VECINV(K,KA)*D00(KA,J,I)
 260  CONTINUE
      SUMF1=0.D0
      SUMF2=0.D0
CCC   DO 270 I=2,MM1
      DO 270 I=ISTART2,IEND1
      DO 270 J=2,NN1
      DO 270 K=1,KM
        F2(K,J,I)=-AM7(K)*PS1(J,I)
        SUMF1=SUMF1+F1(K,J,I)
        SUMF2=SUMF2+F2(K,J,I)
 270  CONTINUE
C   - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
C       Output data for validation
C   - -  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
      IROOT=0
      KOUNT=GCOUNT(MYID)
      CALL MPI_GATHERV (F2, KOUNT, MPI_REAL8,
     1                  TT, GCOUNT, GDISP, MPI_REAL8,
     2                  IROOT, MPI_COMM_WORLD, IERR)
```

```
      CALL MPI_REDUCE (SUMF1, GSUMF1, 1, MPI_REAL8, MPI_SUM,
     1                 IROOT, MPI_COMM_WORLD, IERR)
      CALL MPI_REDUCE (SUMF2, GSUMF2, 1, MPI_REAL8, MPI_SUM,
     1                 IROOT, MPI_COMM_WORLD, IERR)
  301 FORMAT(8E10.3)
      IF (MYID.EQ.0) THEN
        PRINT *,'SUMF1,SUMF2=', GSUMF1,GSUMF2
        PRINT *,' F2(2,2,I),I=1,160,5'
        PRINT 301,(TT(2,2,I),I=1,160,5)
      ENDIF
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,' MYID, CLOCK TIME=', MYID, CLOCK
      CALL MPI_FINALIZE (IERR)
      STOP
      END
```

平行程式 T6DCP 在四個 CPU 上的測試結果如下 ：

```
ATTENTION: 0031-408  8 nodes allocated by LoadLeveler, continuing...
 MYID,NPROC,ISTARTG,IENDG= 0  8  1  20
 MYID,NPROC,ISTARTG,IENDG= 1  8  21  40
 MYID,NPROC,ISTARTG,IENDG= 2  8  41  60
 MYID,NPROC,ISTARTG,IENDG= 4  8  81  100
 MYID,NPROC,ISTARTG,IENDG= 5  8  101  120
 MYID,NPROC,ISTARTG,IENDG= 6  8  121  140
 MYID,NPROC,ISTARTG,IENDG= 7  8  141  160
 MYID,NPROC,ISTARTG,IENDG= 3  8  61  80
 MYID, CLOCK TIME= 1  0.183837374905124307
 MYID, CLOCK TIME= 3  0.184387349989265203
 MYID, CLOCK TIME= 2  0.174135375069454312
 MYID, CLOCK TIME= 6  0.186488100094720721
 MYID, CLOCK TIME= 7  0.184321774868294597
 MYID, CLOCK TIME= 5  0.184246625052765012
 SUMF1,SUMF2= 26172.4605364287745  -2268.89180334311050
 F2(2,2,I),I=1,160,5
  .000E+00 -.333E+00 -.295E+00 -.281E+00 -.274E+00 -.269E+00 -.266E+00 -.264E+00
 -.262E+00 -.261E+00 -.260E+00 -.259E+00 -.258E+00 -.258E+00 -.257E+00 -.257E+00
```

-.256E+00 -.256E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.255E+00 -.254E+00
-.254E+00 -.254E+00 -.254E+00 -.254E+00 -.254E+00 -.253E+00 -.253E+00 -.253E+00
MYID, CLOCK TIME= 0 0.204202200053259730
MYID, CLOCK TIME= 4 0.202977724839001894

在八個 CPU 上執行使用 MPI_SENDRECV 的平行程式 T5DCP 需時 0.0832 秒，在同類電腦八個 CPU 上執行使用 MPI_ISEND、MPI_IRECV 和 MPI_WAIT 的平行程式 T6DCP 卻需時 0.2 秒。使用 Nonblocking 點對點傳送指令反而比使用 Blocking 點對點傳送指令來得慢，資料傳送與計算重疊的效果並沒有顯現出來。

6.2 資料傳送的合併

由於 CPU 之間資料傳送的速度遠低於記憶體內資料移動的速度，若有多筆資料要傳送給同一個 CPU 時，可以使用 MPI_PACK 指令將多筆資料逐一集結在一起，放在一個 BUFFER 裏，當做一筆資料來傳送。對應 CPU 收到這一個 BUFFER 之後，再使用 MPI_UNPACK 指令依序從這一個 BUFFER 裏取得對應每一筆資料。這樣，寫起來是比較麻煩，但是執行起來卻比較快。集結的資料筆數越多，其效果越好。例如下面兩個 MPI_SENDRECV 都是要把陣列的 IEND 項送給右鄰，同時自左鄰收取該陣列的 ISTARTM1 項 ：

```
 ITAG=110
 CALL MPI_SENDRECV (PS1(1,IEND),         N, MPI_REAL8, RIGHT_NBR, ITAG,
1                           PS1(1,ISTARTM1), N, MPI_REAL8,    LEFT_NBR, ITAG,
2                           MPI_COMM_WORLD, ISTATUS, IERR)
 ITAG=120
 CALL MPI_SENDRECV (PS2(1,IEND),         N, MPI_REAL8, RIGHT_NBR, ITAG,
1                           PS2(1,ISTARTM1), N, MPI_REAL8,    LEFT_NBR, ITAG,
2                           MPI_COMM_WORLD, ISTATUS, IERR)
```

可以使用兩個 MPI_PACK 指令把這兩筆 IEND 陣列資料集結在字符陣列 BUF1 裏，然後使用一個 MPI_SENDRECV 來把 BUF1 送給 RIGHT_NBR，同時自左鄰收取得 BUF2，最後再使用兩個 MPI_UNPACK 指令從 BUF2 取得各個對應陣列的 ISTARTM1 資料項目如下。

```
INTEGER       BUFSIZE
PARAMETER (BUFSIZE=N*2*8)
CHARACTER BUF1(BUFSIZE),BUF2(BUFSIZE)
```

```
 CALL MPI_BARRIER (MPI_COMM_WORLD,IERR)
 IPOS=0
 CALL MPI_PACK ( PS1(1,IEND), N,    MPI_REAL8, BUF1, BUFSIZE,
&                IPOS, MPI_COMM_WORLD, IERR)
 CALL MPI_PACK ( PS2(1,IEND), N,    MPI_REAL8, BUF1, BUFSIZE,
&                IPOS, MPI_COMM_WORLD, IERR)
 KOUNT=N*2*8
 ITAG=120
 CALL MPI_SENDRECV (BUF1, KOUNT, MPI_CHARACTER, R_NBR, ITAG,
1                   BUF2, KOUNT, MPI_CHARACTER, L_NBR, ITAG,
2                   MPI_COMM_WORLD, ISTATUS, IERR)
 IF (MYID.NE.0) THEN
   IPOS=0
   CALL MPI_UNPACK ( BUF2, KOUNT, IPOS, PS1(1,ISTARTM1), N,
1                   MPI_REAL8,    MPI_COMM_WORLD, IERR)
   CALL MPI_UNPACK ( BUF2, KOUNT, IPOS, PS2(1,ISTARTM1), N,
1                   MPI_REAL8,    MPI_COMM_WORLD, IERR)
 ENDIF
```

不採用 PACK、UNPACK 的指令時，也可以採用下述方式來完成相同的工作。

```
 REAL*8    BUF1(N,2), BUF2(N,2)
 DO J=1,N
   BUF1(J,1)=PS1(J,IEND)
   BUF1(J,2)=PS2(J,IEND)
 ENDDO
 KOUNT=N*2
 ITAG=120
 CALL MPI_SENDRECV (BUF1, KOUNT, MPI_REAL8, RIGHT_NBR, ITAG,
1                   BUF2, KOUNT, MPI_REAL8,    LEFT_NBR, ITAG,
2                   MPI_COMM_WORLD, STATUS, IERR)
 IF (MYID.NE.0) THEN
   DO J=1,N
     PS1(J,ISTARTM1)=BUF2(J,1)
     PS2(J,ISTARTM1)=BUF2(J,2)
   ENDDO
 ENDIF
```

這樣一來，原來需要兩個 MPI_SENDRECV 指令，現在只需要一個 MPI_SENDRECV 指令，原來傳送的資料長度為 N，現在則為兩筆資料長度的加總。圖 6.3 和圖 6.4 是國家高速電腦中心的 IBM SP2、IBM SP2 SMP、HP SPP2200、和 Fujitsu VPP300 四種機型上兩個同質 CPU 之間傳送各種不同長度資料的速度圖。由該圖可以看出傳送的資料長度在 1 MB 以上其傳送速度漸趨於飽和，傳送的資料長度在 1 MB 以下部份，則是傳送的資料越長其傳送速度越快。

另外，還可以使用導出的資料類型 MPI_TYPE_STRUCT 來達到多筆傳送資料合併的目的，請參閱 7.1 節。

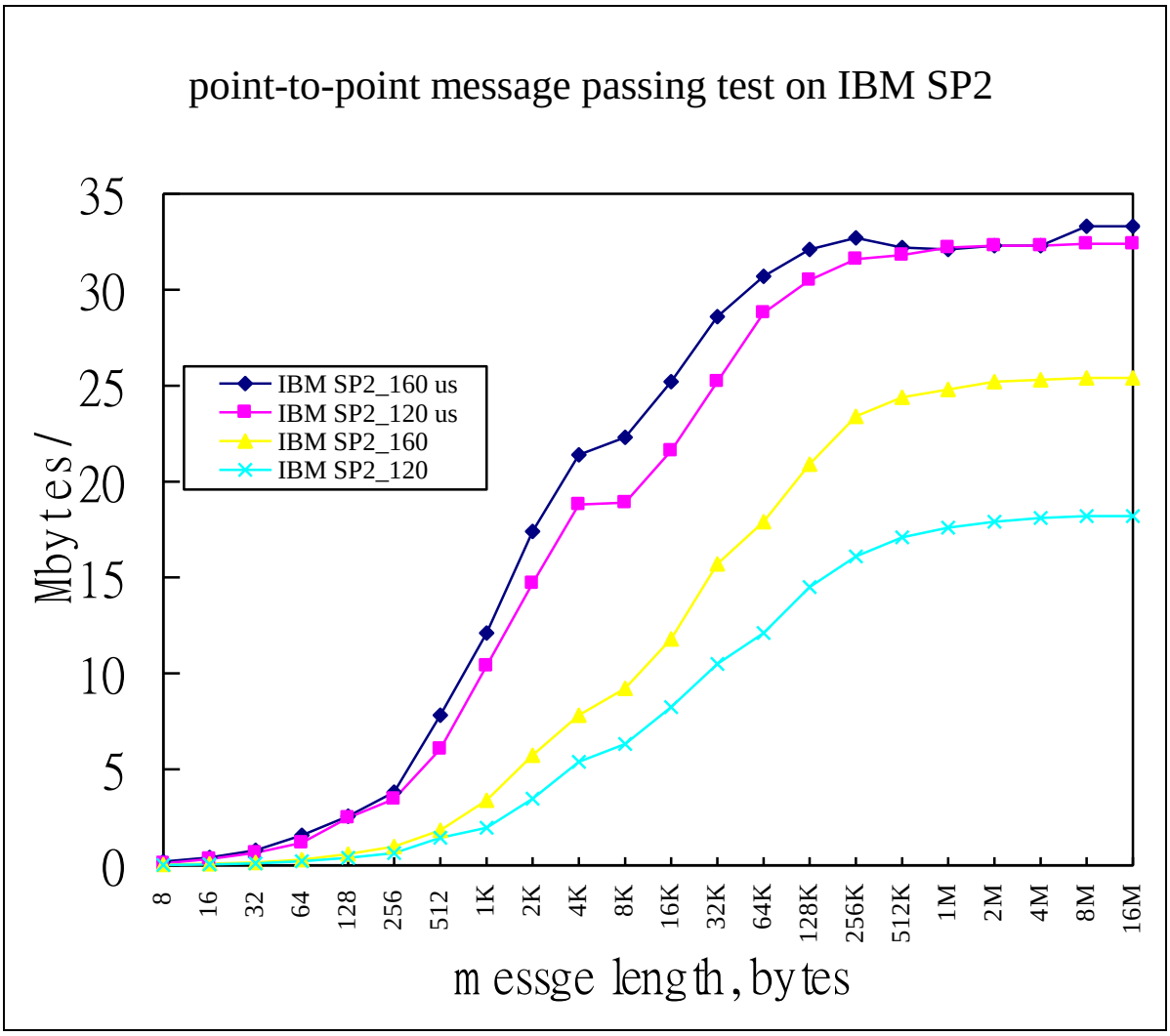


圖 6.3　IBM SP2 同一種 CPU 間點對點的傳送速度圖

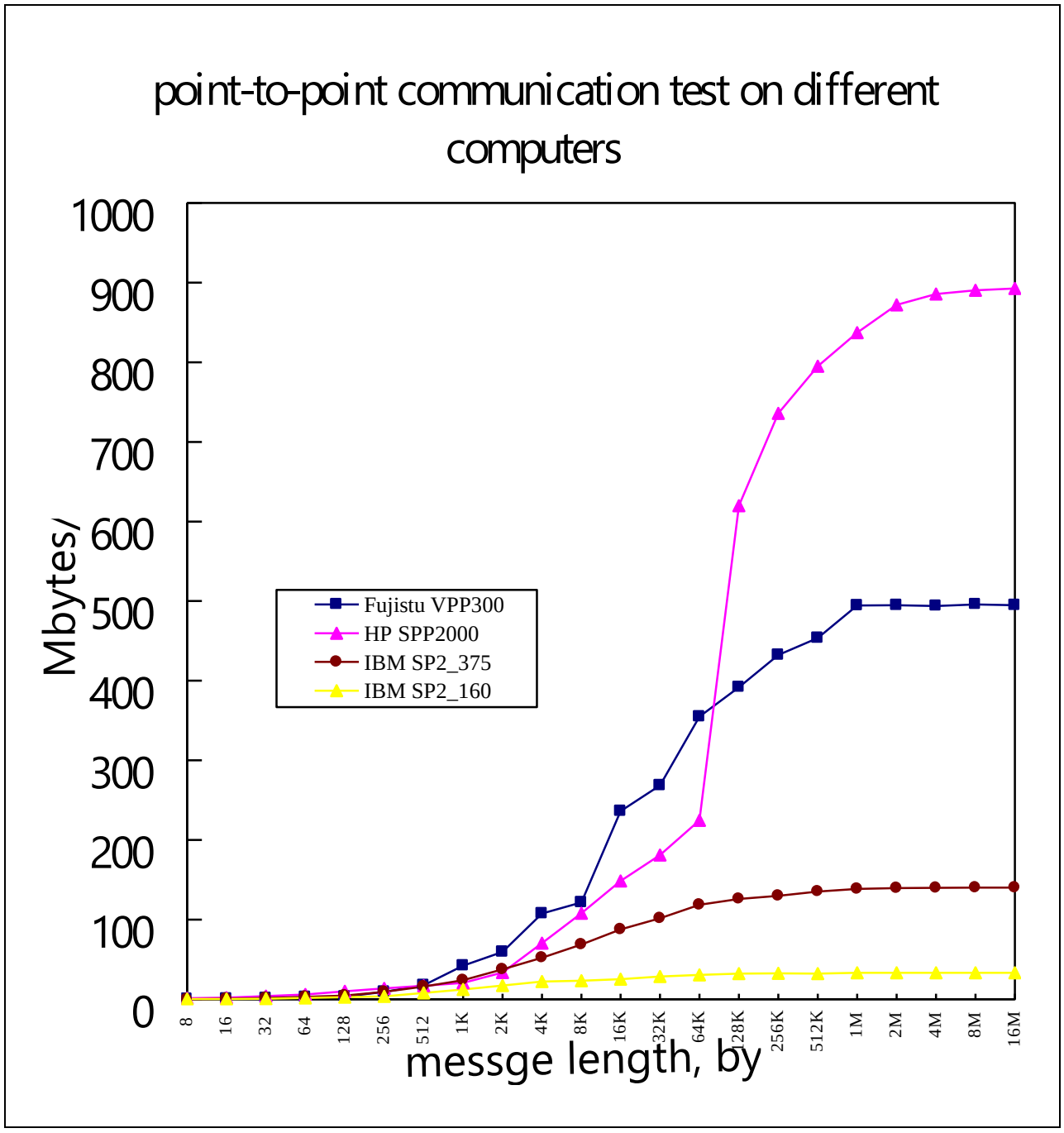


圖 6.4 各個廠牌電腦系統同質 CPU 間點對點的傳送速度圖

6.3 以邊界資料計算取代邊界資料交換

由於 CPU 之間資料傳送的速度遠低於 CPU 的計算速度，如果由已經備妥邊界資料的陣列導出的陣列需要作邊界資料交換時，可以用邊界資料計算取代邊界資料交換，來減少邊界資料交換的次數，以提高平行計算效率。

例如在下面這一段程式裏，如果在 loop-30 之前已經備妥 PS2 及 U2 的 I-1、I+1 邊界資料時，在 loop-30 裏由 PS2 導出的陣列 D1 在 loop-40 裏要用到 I-1 的邊界資料，因此必需在 loop-40 之前交換 D1 的 I-1 邊界資料。

```
CCC    DO 30 I=1,M1
       DO 30 I=ISTART,IEND1
       DO 30 J=1,N1
          D1(J,I)=(PS2(J,I+1)+PS2(J,I))*HXU(J,I)*0.5D0
          D2(J,I)=(PS2(J+1,I)+PS2(J,I))*HXV(J,I)*0.5D0
  30   CONTINUE
       CALL MPI_SENDRECV(D1(1,IEND),        NX, MPI_REAL8, R_NBR, ITAG,
      1                  D1(1,ISTARTM1), NX, MPI_REAL8, L_NBR, ITAG,
      2                  MPI_COMM_WORLD, ISTATUS, IERR)
CCC    DO 40 I=2,M1
       DO 40 I=ISTART2,IEND1
       DO 40 K=1,KK
       DO 40 J=2,N1
          DUMMY1(J,K,I)=
      1      (D1(J,I)*U2(J,K,I)-D1(J,I-1)*U2(J,K,I-1))*HMMX(J,I)+
      2      (D2(J,I)*V2(J,K,I)-D2(J-1,I)*V2(J-1,K,I))*HMMY(J,I)
  40   CONTINUE
```

既然在 loop-30 之前已經備妥 PS2 及 U2 的 I-1、I+1 邊界資料，就可以在 loop-30 裏多算 I-1 的邊界資料，用來取代 loop-30 之後陣列 D1 的邊界資料交換。如下

```
CCC    DO 30 I=1,M1
CCC    DO 30 I=ISTART,IEND1
       DO 30 I=ISTARTM1,IEND1
       DO 30 J=1,N1
         D1(J,I)=(PS2(J,I+1)+PS2(J,I))*HXU(J,I)*0.5D0
         D2(J,I)=(PS2(J+1,I)+PS2(J,I))*HXV(J,I)*0.5D0
  30   CONTINUE
CCC    DO 40 I=2,M1
       DO 40 I=ISTART2,IEND1
       DO 40 K=1,KK
       DO 40 J=2,N1
         DUMMY1(J,K,I)=
     1       (D1(J,I)*U2(J,K,I)-D1(J,I-1)*U2(J,K,I-1))*HMMX(J,I)+
     2       (D2(J,I)*V2(J,K,I)-D2(J-1,I)*V2(J-1,K,I))*HMMY(J,I)
  40   CONTINUE
```

如此，就可以減少邊界資料交換的次數，達到提高平形計算效率的目的。

例如氣象局的 RFS 程式，就選擇副程式 CHEF 在更新過陣列 U1、V1、T1、Q1、PS1、U3、V3、T3、Q3、PS3、W、P1 之後，使用 MPI_PACK、MPI_UNPACK 集結多筆資料做邊界資料交換。之後，由這些已經備妥邊界資料的陣列導出的陣列需要作邊界資料交換時，就可以用邊界資料計算取代邊界資料交換，減少了許多邊界資料交換的次數，提高了平行計算效率好幾個百分點。

6.4 輸出入資料的安排

在 MPI 能提供平行輸出入功能之前，輸入資料的處理方式除了由某一個 CPU 將資料讀進來之後使用 MPI_SCATTER、MPI_SCATTERV 或 MPI_BCAST 分送給各個 CPU 之外，還可以事先將切割好的陣列片段寫到公用磁碟的各個磁檔上，平行程式的各個 CPU 分別從預定的磁檔讀入該 CPU 所需要的資料。

輸出資料的處理方式除了使用 MPI_GATHER、MPI_GATHERV 來收集各個 CPU 的計算結果之外，還可以由各個 CPU 把該 CPU 轄區裏的資料各別寫到各自的檔案裏，然後寫一個小程式來收集這些檔案裏的資料，集結成完整的資料檔案。

6.4.1 事先切割輸入資料

例如，下列程式片段是把一維陣列 B、C、D 讀進來之後，切割為 NP 段，分別寫到檔名為 input.11、input.12、input.13、 . . .等磁檔上

```
      CHARACTER   FNAME*8
      REAL*8          A(MM), B(MM), C(MM), D(MM)
      NP=4
      OPEN(7, FILE='input.dat', FORM='UNFORMATTED')
      READ (7) B
      READ (7) C
      READ (7) D
      CLOSE (7)
  101 FORMAT('input.',I2)
      DO I=0,NP-1
        IU=11+I
        WRITE(FNAME,101) IU
        OPEN(IU,FILE=FNAME,FORM='UNFORMATTED')
        CALL STARTEND(I, NP, 1, MM, ISTART, IEND)
```

```
      WRITE(IU) (B(I), I=ISTART, IEND)
      WRITE(IU) (C(I), I=ISTART, IEND)
      WRITE(IU) (D(I), I=ISTART, IEND)
      CLOSE(IU)
    ENDDO
```

如果是在第三維切割的三維陣列，則寫出陳述改為

```
      WRITE(IU) (((B(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
      WRITE(IU) (((C(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
      WRITE(IU) (((D(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
```

平行程式依相同的檔案命名方式從公用磁檔讀入該 CPU 所需要的資料如下。

```
      PARAMETER (MM=200, NP=4, M=MM/NP)
      INCLUDE       'mpif.h'
      REAL*8          A(0:M+1), B(0:M+1), C(0:M+1), D(0:M+1)
      INTEGER         NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
     1                 LEFT_NBR, RIGHT_NBR
      CHARACTER    FNAME*8

      CALL MPI_INIT (MPIERROR)
      CALL MPI_COMM_SIZE    (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD,    MYID, IERR)
      ISTART=1
      IEND=M
      . . . . . .
101   FORMAT ('input.',I2)
      IU=MYID+11
      WRITE(FNAME,101) IU
      OPEN(IU,FILE=FNAME,STATUS='OLD',FORM='UNFORMATTED')
      READ (IU)    (B(I), I=ISTART, IEND)
      READ (IU)    (C(I), I=ISTART, IEND)
      READ (IU)    (D(I), I=ISTART, IEND)
```

如果是在第三維切割的三維陣列，則讀入陳述改為

```
READ (IU) (((B(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
READ (IU) (((C(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
READ (IU) (((D(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
```

採用這種輸入資料的處理方式，雖然需要多寫一個小程式來切割輸入資料，對平行程式來說寫起來比較簡單，每個 CPU 讀入的資料量相差無幾，有益於各個 CPU 的負載平衡(load balance)。當然，如果這些檔案是存放在同一棵磁碟裏，多棵 CPU 同時讀資料時還是必須輪流進行，等待在所難免。如果能夠事先把這些檔案從公用磁碟抄到各個 CPU 的私有磁碟(local disk)裏，則平行程式各個 CPU 都是從該 CPU 的私有磁碟讀入資料，就不必等待而達到更理想的負載平衡。各個 CPU 的私有磁碟暫用檔案名稱必須請教該平行系統管理人員。

下面的程式片段是在平行程式裏叫用 SYSTEM 副程式把該 CPU 需要的輸入檔案 input.xx 從公用磁碟抄到該 CPU 的私有磁碟 /var/tmp 裏，然後從該 CPU 的 /var/tmp 裏讀入檔案 input.xx。

```
 PARAMETER (NTOTAL=200,NP=4,N=NTOTAL/NP)
 INCLUDE 'mpif.h'
 REAL*8     A(0:N+1),B(0:N+1),C(0:N+1),D(0:N+1), T(NTOTAL),
&           AMAX,GMAX
 CHARACTER FNAME*30, CMD*30

 INTEGER   NPROC,MYID,ISTATUS(MPI_STATUS_SIZE),ISTART,IEND,
1           LEFT_NBR,RIGHT_NBR
```

```
      CALL MPI_INIT(MPIERROR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROC,IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYID, IERR)
      .  .  .  .  .  .
101   FORMAT ('cp input.',I2,'    /var/tmp')
102   FORMAT ('/var/tmp/input.',I2)
      IU=11+MYID
      WRITE(CMD,101) IU
      CALL SYSTEM (CMD)
      WRITE (FNAME,102) IU
      OPEN(IU, FILE=FNAME, STATUS='OLD', FORM='UNFORMATTED')
      READ (IU)    (B(I), I=ISTART, IEND)
      READ (IU)    (C(I), I=ISTART, IEND)
      READ (IU)    (D(I), I=ISTART, IEND)
      CLOSE (IU, STATUS=’DELETE’)
```

6.4.2 事後收集切割過的輸出資料

平行程式的檔案輸出也可以採用同樣的方式來處理。每個 CPU 寫出該 CPU 轄區內的資料到各別的檔案裏，然後寫一個小程式來收集這些檔案裏的資料，最後再寫成一個完整的檔案。例如在平行程式裏把計算出來的 A 陣列片段寫到 output.xx 檔裏如下

```
      PARAMETER (MM=200, NP=4, M=MM/NP)
      INCLUDE       'mpif.h'
      REAL*8          A(0:M+1), B(0:M+1), C(0:M+1), D(0:M+1)
      INTEGER       NPROC, MYID, ISTATUS(MPI_STATUS_SIZE), ISTART, IEND,
103   LEFT_NBR, RIGHT_NBR
      CHARACTER    FNAME*8

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
```

```
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      ISTART=1
      IEND=M
      . . . . . .
101   FORMAT ('output.',I2)
      IU=MYID+11
      WRITE (FNAME,101)    IU
      OPEN (IU, FILE=FNAME, STATUS='NEW', FORM='UNFORMATTED')
      WRITE (IU) (A(I), I=ISTART, IEND)
      .  .  .  .  .  .  .
```

如果是在第三維切割的三維陣列，則寫出陳述改為

```
      WRITE(IU) (((A(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
```

下面的程式片段是從 output.11、output.12、output.13、 . . .等 NP 個磁檔讀入陣列的部分資料，組合成一個完整的陣列資料，再寫出去。

```
      CHARACTER    FNAME*8
      REAL*8           A(MM), B(MM), C(MM), D(MM)
      NP=4
101   FORMAT('output.',I2)
      DO I=0,NP-1
        IU=11+I
        WRITE (FNAME,101) IU
        OPEN (IU, FILE=FNAME, STATUS='OLD', FORM='UNFORMATTED')
        CALL STARTEND (I, NP, 1, MM, ISTART, IEND)
        READ (IU)    (A(I), I=ISTART, IEND)
```

```
    CLOSE (IU)
ENDDO
OPEN (7,FILE='output', STATUS='new', FORM='unformatted')
WRITE (7)    A
CLOSE (7)
.  .  .  .  .  .  .  .  .
```

如果是在第三維切割的三維陣列，則讀入陳述改為

```
READ (IU) (((A(K,J,I), K=1,L), J=1,N), I=ISTART,IEND)
```

第七章　導出的資料類別

本章將介紹 MPI 導出的資料類別、陣列的轉換 (Transposing Block Distribution)、及兩方迴歸與管線法 (2 Way Recursive and Pipeline method)。

第一節 導出的資料類別。

第二節 陣列的轉換。

第三節 兩方迴歸與管線法。

7.1 導出的資料類別

MPI 的資料類別除了基本的資料類別如 MPI_INTEGER、MPI_REAL、MPI_REAL8、MPI_CHARACTER、MPI_LOGICAL 等等之外，還備有 “導出的資料類別” (derived data type)，讓程式設計師自己來設定他所需要的資料類別，包括 MPI_TYPE_VECTOR、MPI_TYPE_CONTIGUOUS、MPI_TYPE_INDEXED、MPI_TYPE_STRUCT 等指令。其中的 MPI_TYPE_VECTOR 是用來設定一個陣列裏 “固定間隔” (Constant Stride)的資料串，在第五章裏已經介紹過。MPI_TYPE_CONTIGUOUS 可以用來設定陣列裏連續位址的一串資料為一個資料類別，MPI_TYPE_STRUCT 可以用來設定多種資料類別組合起來的集合體、或多個單一資料類別組合起來的集合體，類似 C 語言的 struct 或 Fortran90 的 TYPE 指令。

例如, 在 Fortran90 程式裏可以經由下列 TYPE 指令將兩個實數與一個整數的集合體當做一種資料類別來看待。

```
TYPE   LOAD
  REAL   A
  REAL   B
  INTERGER   N
END TYPE   LOAD
```

在 MPI 程式裏則要用 MPI_TYPE_STRUCT 來加以設定，其叫用格式如下 ：

```
CALL MPI_TYPE_STRUCT( COUNT, LENGTH, DISP, OLDTYPE, NEWTYPE, IERR)
CALL MPI_TYPE_COMMIT( NEWTYPE, IERR)
```

其引數

COUNT　　為集合體裏的資料項數，整常數或整變數

LENGTH　　每項資料的長度(個數) ，整數陣列

DISP　　每項資料起點的相對位址，整數陣列

OLDTYPE　　每項資料原來的資料類別，整數陣列

NEWTYPE　　新設定的資料類別，整變數

集合體裏各項資料的位址可以叫用 MPI_ADDRESS 來取得，然後再計算其相對位址 (Displacement)。其叫用格式如下 ：

CALL MPI_ADDRESS(DATA, ADDRESS, IERR)

其引數

DATA　　資料的起點

ADDRESS　　DATA 起點之位址，整數變數

則上述 Fortran 90 之集合體在 Fortran 90 程式裏以 MODULE 型態來設定後，任何副程度需要用到該集合體時只要在其宣告陳述之前 USE 該 MODULE 名稱即可。舉例如下 ：

```
MODULE    MLOAD
    TYPE LOAD
      REAL    A
      REAL    B
      INTEGER    N
    END TYPE LOAD
END MODULE    MLOAD
```

```
C-----------------------------------------------------------------------
      PROGRAM    STRUCT
      USE            MLOAD
      INCLUDE    'mpif.h'
      INTEGER    IBLOCK(3), IDISP(3), ITYPE(3)
      INTEGER    ISTATUS(MPI_STATUS_SIZE)
      TYPE(LOAD)    ABN

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)

      IBLOCK(1)=1
      IBLOCK(2)=1
      IBLOCK(3)=1
      ITYPE(1)=MPI_REAL
      ITYPE(2)=MPI_REAL
      ITYPE(3)=MPI_INTEGER
      CALL MPI_ADDRESS ( ABN%A, IDISP(1), IERR)
      CALL MPI_ADDRESS ( ABN%B, IDISP (2), IERR )
      CALL MPI_ADDRESS ( ABN%N, IDISP (3), IERR )
      DO I=3,1,-1
        IDISP(I)=IDISP(I)-IDISP(1)
      ENDDO
      CALL MPI_TYPE_STRUCT(3, IBLOCK, IDISP, ITYPE, LLOAD, IERR)
      CALL MPI_TYPE_COMMIT( LLOAD, IERR)
      ITAG=9
      IF(MYID.EQ.0) THEN
        READ(*,*)    ABN
        CALL MPI_SEND(ABN, 1, LLOAD, 1, ITAG, MPI_COMM_WORLD, IERR)
      ELSE
        CALL MPI_RECV(ABN, 1, LLOAD, 0, ITAG, MPI_COMM_WORLD,
     &                          ISTATUS, IERR)
        PRINT *,'ABN =',    ABN
      ENDIF
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

輸入資料之內容為 ：

5.0　8.0　12

則在兩棵 CPU 上平行執行的結果如下

```
ATTENTION: 0031-408    2 tasks allocated by LoadLeveler, continuing...
 ABN = 5.000000000    8.000000000    12
```

由此可知，資料輸出入時要使用 Fortran90 設定的資料名稱，而 CPU 間資料傳送時要使用 Fortran90 設定的資料名稱及 MPI_TYPE_STRUCT 所設定的資料類別。

下列程式片段是在二維陣列的第二維做平行記算時，要把該 CPU 最左邊的一列或兩列資料傳送給左鄰，同時從右鄰取得右邊界外的一列或兩列資料。此處有三個陣列資料要傳送，為了提高傳送效率，捨棄三個 MPI_SENDRECV 指令而改用三個 MPI_PACK 指令把三段資料集結到 BUF1 裏，使用一個 MPI_SENDRECV 指令來送出 BUF1 及收入 BUF2 資料。收到資料後再用三個 MPI_UNPACK 指令從 BUF2 依序取出收到的資料。

```
      REAL              UP(KM,IMP1), VP(KM,IMP1), WP(KM,IMP1)
      INTEGER          BUFSIZE
      PARAMETER    (BUFSIZE=KM*4*4)
      CHARACTER    BUF1(BUFSIZE),    BUF2(BUFSIZE)
      .  .  .  .  .  .  .  .  .  .  .  .  .
      KM2=KM*2
      IF(MYID.NE.0) THEN
        IPOS=0
        CALL MPI_PACK ( UP(1,ISTART), KM, MPI_REAL, BUF1, BUFSIZE, IPOS,
     1                          MPI_COMM_WORLD, IERR)
```

```
      CALL MPI_PACK ( VP(1,ISTART), KM2, MPI_REAL, BUF1, BUFSIZE, IPOS,
     1                MPI_COMM_WORLD, IERR)
      CALL MPI_PACK ( WP(1,ISTART), KM, IREAL, BUF1, BUFSIZE, IPOS,
     1                MPI_COMM_WORLD, MPI_ERR)
    ENDIF
    ITAG=202
    CALL MPI_SENDRECV( BUF1, BUFSIZE, MPI_CHARACTER,   L_NBR, ITAG,
     1                 BUF2, BUFSIZE, MPI_CHARACTER,   R_NBR, ITAG,
     2                 MPI_COMM_WORLD, ISTATUS, IERR)
    IF(MYID.NE.NPROC-1) THEN
      IPOS=0
      CALL MPI_UNPACK (BUF2, BUFSIZE, IPOS, UP(1,IENDP1), KM, MPI_REAL,
     1                 MPI_COMM_WORLD, IERR)
      CALL MPI_UNPACK (BUF2, BUFSIZE, IPOS, VP(1,IENDP1), KM2, MPI_REAL,
     1                 MPI_COMM_WORLD, IERR)
      CALL MPI_UNPACK (BUF2, BUFSIZE, IPOS,WP(1,IENDP1), KM, MPI_REAL,
     1                 MPI_COMM_WORLD, IERR)
    ENDIF
```

此處，待傳送資料的數量單位是 KM 個連續位址的資料串，可以用 MPI_TYPE_CONTIGUOUS 來設定其資料類別，取名 KMREAL :

```
      CALL MPI_TYPE_CONTIGUOUS ( KM, MPI_REAL, KMREAL, IERR)
      CALL MPI_TYPE_COMMIT (KMREAL, IERR)
```

整個程式都會用到的導出資料類別之設定，在 Fortran 77 程式最好是在主程式裏一次設定，並且放在 COMMON/MPIBLK/ 公用區段裏，各個副程式隨時可以取用。在 Fortran 90 程式則宜寫成 MODULE，要用到它的副程式 USE 該 MODULE 即可。至於副程式裏各別要用到的導出資料類別，可在各該副程式裏分別設定。由於副程式往往會被叫用多次，為了避免因多次設定而浪費時間，可以設法只在第一次被叫用時設定。

此處要交換的邊界資料一共有三段，UP 陣列上是 KM 個元素、VP 陣列上是 KM*2 個元素、WP 陣列上是 KM 個元素，送出資料的起點都是(1,ISTART)，收入資料的起點都是(1,IENDP1) ，而且 UP、VP、WP 三個陣列的 dimension 也完全一樣。我們可以使用下述程式片刻來設定導出資料類別 IPACK3，來描述要交換的邊界資料後，叫用 MPI_SENDRECV 來傳送這一種資料類別，取代上面這一段 PACK 和 UNPACK 的工作。

```
      INTEGER    IBLOCK(3), ITYPE(3), IDISP(3), IPACK3, IFIRST1
      DATA           IFIRST1/1/
      .   .   .   .   .   .   .
      IF(IFIRST1.EQ.1) THEN
         IFIRST1=0
         IBLOCK(1)=1
         IBLOCK(2)=2
         IBLOCK(3)=1
         ITYPE(1)=KMREAL
         ITYPE(2)=KMREAL
         ITYPE(3)=KMREAL
         CALL MPI_ADDRESS( UP(1,ISTART), IDISP(1), IERR )
         CALL MPI_ADDRESS( VP(1,ISTART), IDISP(2), IERR )
         CALL MPI_ADDRESS( WP(1,ISTART), IDISP(3), IERR )
         DO I=3,1,-1
            IDISP(I)=IDISP(I)-IDISP(1)
         ENDDO
         CALL MPI_TYPE_STRUCT( 3, IBLOCK, IDISP, ITYPE, IPACK3, IERR)
         CALL MPI_TYPE_COMMIT(IPACK3, IERR)
      ENDIF
      ITAG=202
      CALL MPI_SENDRECV( UP(1,ISTART), 1, IPACK3,    L_NBR, ITAG,
     1                                UP(1,IENDP1), 1, IPACK3,    R_NBR, ITAG,
     2                                MPI_COMM_WORLD, ISTATUS, IERR)
```

使用這一種導出的資料類別比使用 PACK/UNPACK 效率較佳，值得多加利用。

7.2 陣列的轉換

如果在二維陣列的計算過程中需要依第二維切割變換為第一維切割，然後又回到第二維切割，兩種切割交互進行時就要用到陣列的轉換(Array Transpose)。圖 7.1 是從第二維切割變換為第一維切割的資料流程。

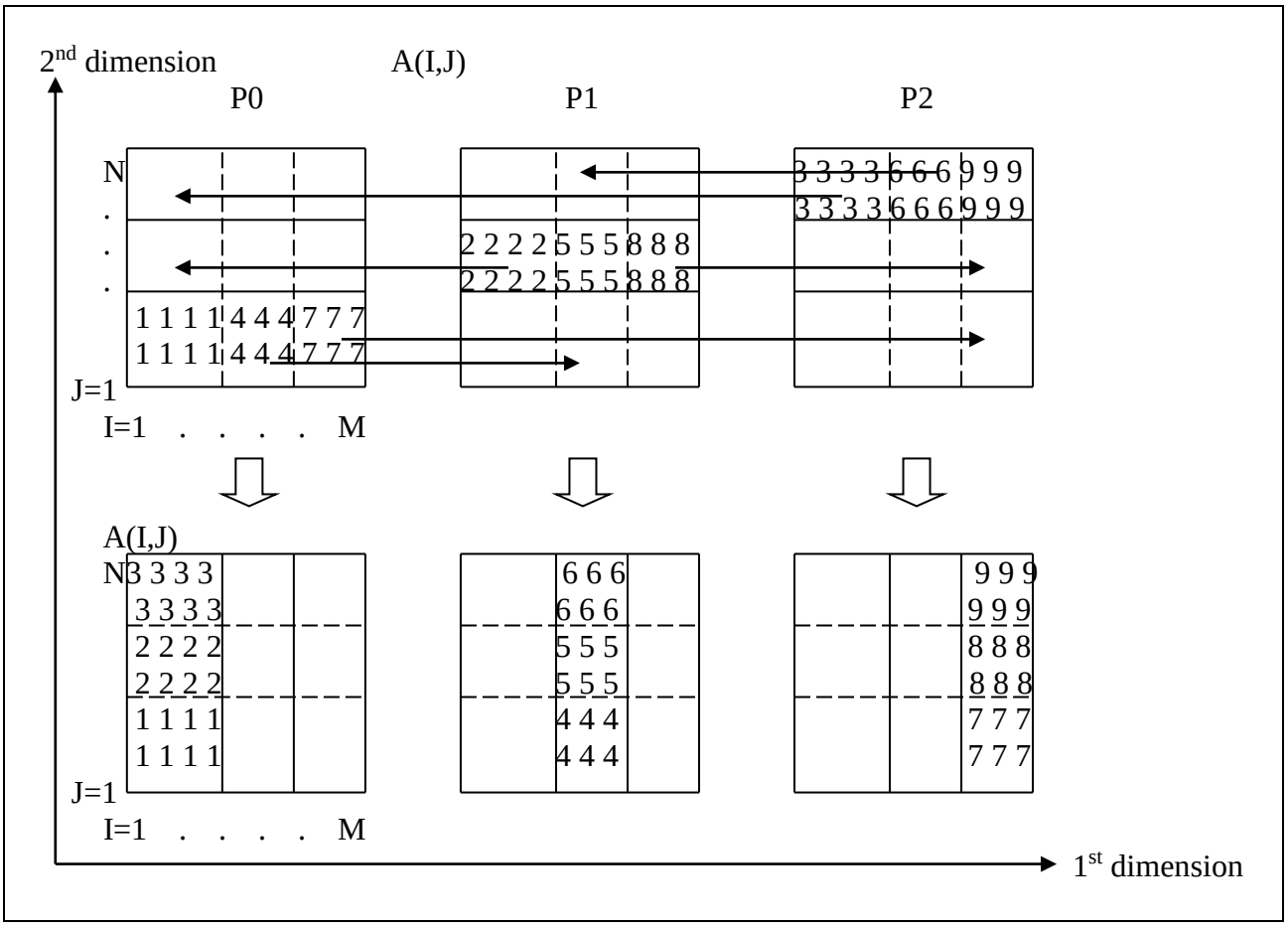


圖 7.1 row_to_col block transpose

如圖 7.1 在三個 CPU 上平行計算時將陣列劃分為九個區塊(4 個 CPU 16 個區塊，5 個 CPU 25 個區塊，餘類推)，CPU 間資料的傳送以這些區塊為單位較為容易。圖 7.1 的上方為第二維的

切割（row distribution），依劍線的方式來傳送資料區塊後，如該圖的下方，就可以依第一維的切割方式（column distribution）來處理。由於格點數不能被 CPU 數整除時區塊的大小不盡相等，使用 derived data type 來設定各個區塊的資料類別較為容易，如圖 7.2 所示。如果(I,J)兩個方向的格點數都能夠被 CPU 數整除時，所有的區塊大小都完全一樣，這時只要設定一種資料類別就足夠了(採用 type_block2a)。

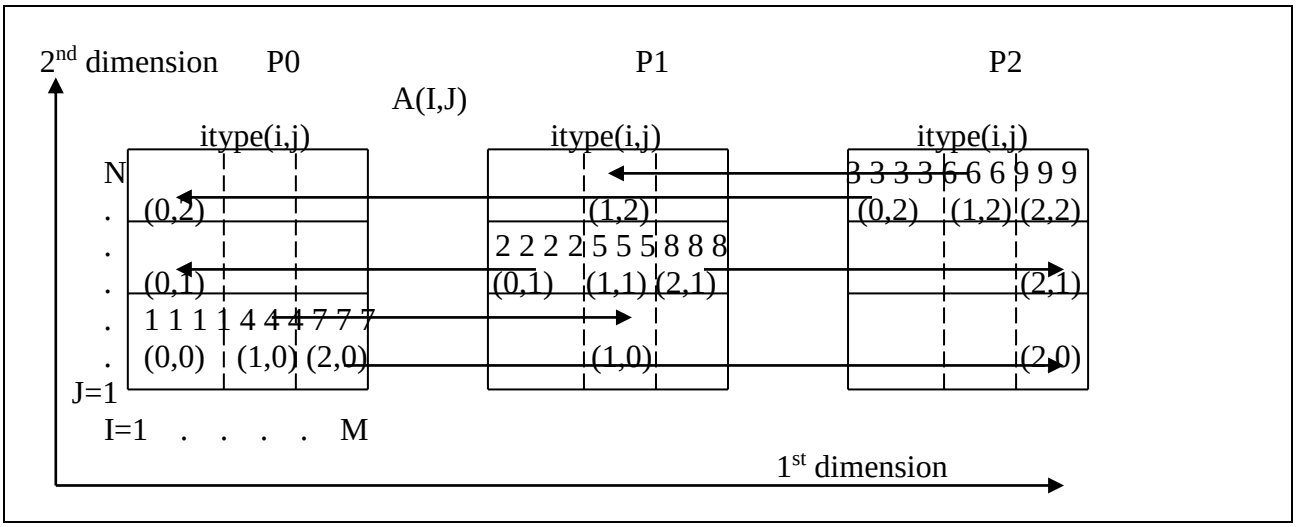
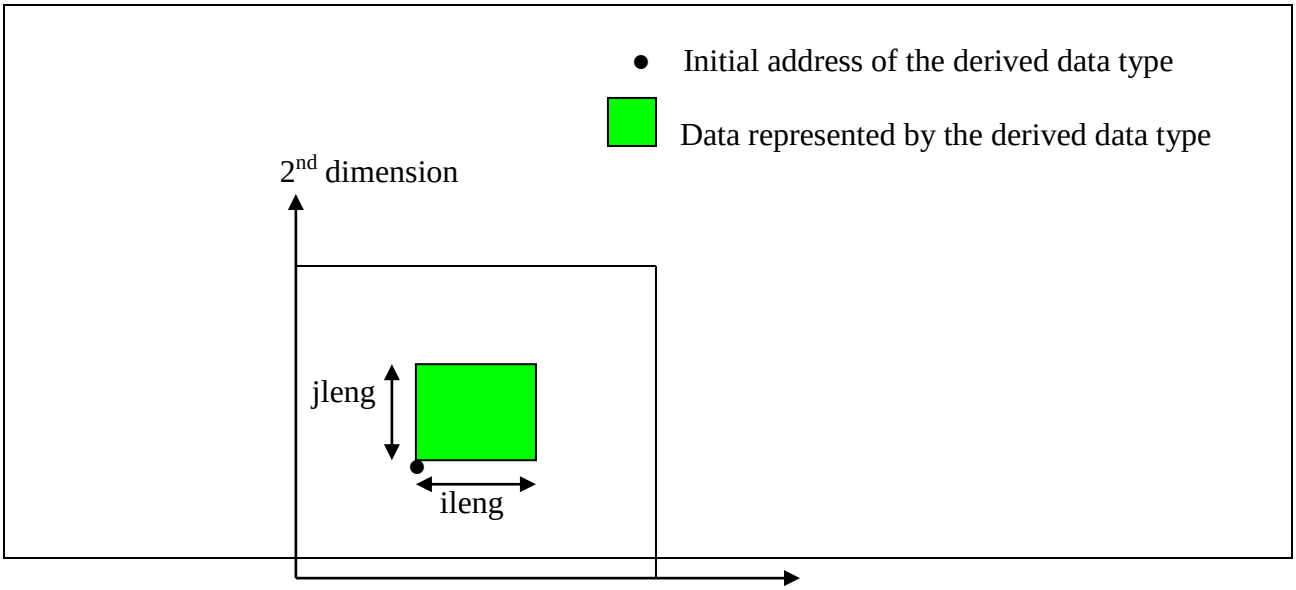


圖 7.2 Derived Data Type 的設定及 Row_to_Col Transpose



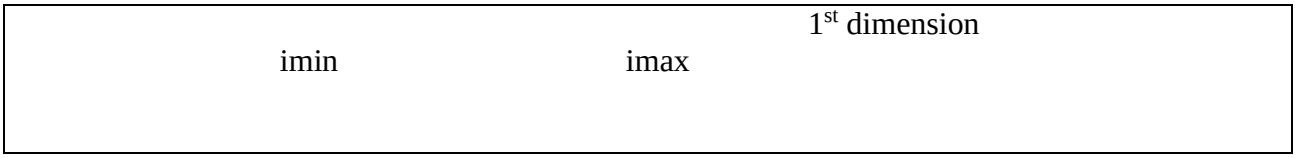


圖 7.4　type_block2a　derived data type

這一種資料區塊的設定方式是以該資料區塊的起點為導出資料類別的起點，如圖 7.4 所示。

這一種資料類別取名 BLOCK2D，這一種資料類別適用於陣列沒有切割以及陣列切割的情況。

```
SUBROUTINE   BLOCK2D(IMIN,IMAX,ILENG,JLENG,IOLDTYPE, NEWTYPE)
INCLUDE      'mpif.h'
INTEGER      IMIN, IMAX, ILENG, JLENG, IOLDTYPE, NEWTYPE
ISTRIDE = IMAX - IMIN +1
CALL MPI_TYPE_VECTOR (JLENG,ILENG,ISTRIDE,IOLDTYPE, NEWTYPE,IERR)
CALL MPI_TYPE_COMMIT (NEWTYPE,IERR)
END
```

使用 BLOCK2D 資料類別　ROW_TO_COL　transpose 之程式片段如下

```
ITAG=10
DO ID=0, NPROC-1
  IF (ID.NE.MYID) THEN
    ISTART1=ISTARTG(ID)
    JSTART1=JSTARTG(ID)
    CALL MPI_ISEND( A(ISTART1,JSTART), 1, ITYPE(ID,MYID), ID, ITAG,
1                   MPI_COMM_WORLD, IREQ1(ID), MPI_ERR)
    CALL MPI_IRECV( A(ISTART,JSTART1), 1, ITYPE(MYID,ID), ID, ITAG,
2                   MPI_COMM_WORLD, IREQ2(ID), MPI_ERR)
  ENDIF
ENDDO
DO ID=0, NPROC-1
  IF(ID.NE.MYID) THEN
    CALL MPI_WAIT (IREQ1(ID), ISTATUS, MPI_ERR)
    CALL MPI_WAIT (IREQ2(ID), ISTATUS, MPI_ERR)
```

```
    ENDIF
  ENDDO
```

以上這一段 non-blocking send/recv 也可以改用 blocked sendrecv 來傳送如下，但是其傳送效率較差，還是採用 non-blocking send/recv 為佳。

```
  DO ID=0, NPROC-1
    IF (ID.NE.MYID) THEN
      ISTART1=ISTARTG(ID)
      JSTART1=JSTARTG(ID)
     CALL MPI_SENDRECV( A(ISTART1,JSTART), 1, ITYPE(ID,MYID), ID, ITAG,
1                       A(ISTART,JSTART1), 1, ITYPE(MYID,ID), ID, ITAG,
2                       MPI_COMM_WORLD, ISTATUS, MPI_ERR)
    ENDIF
  ENDDO
```

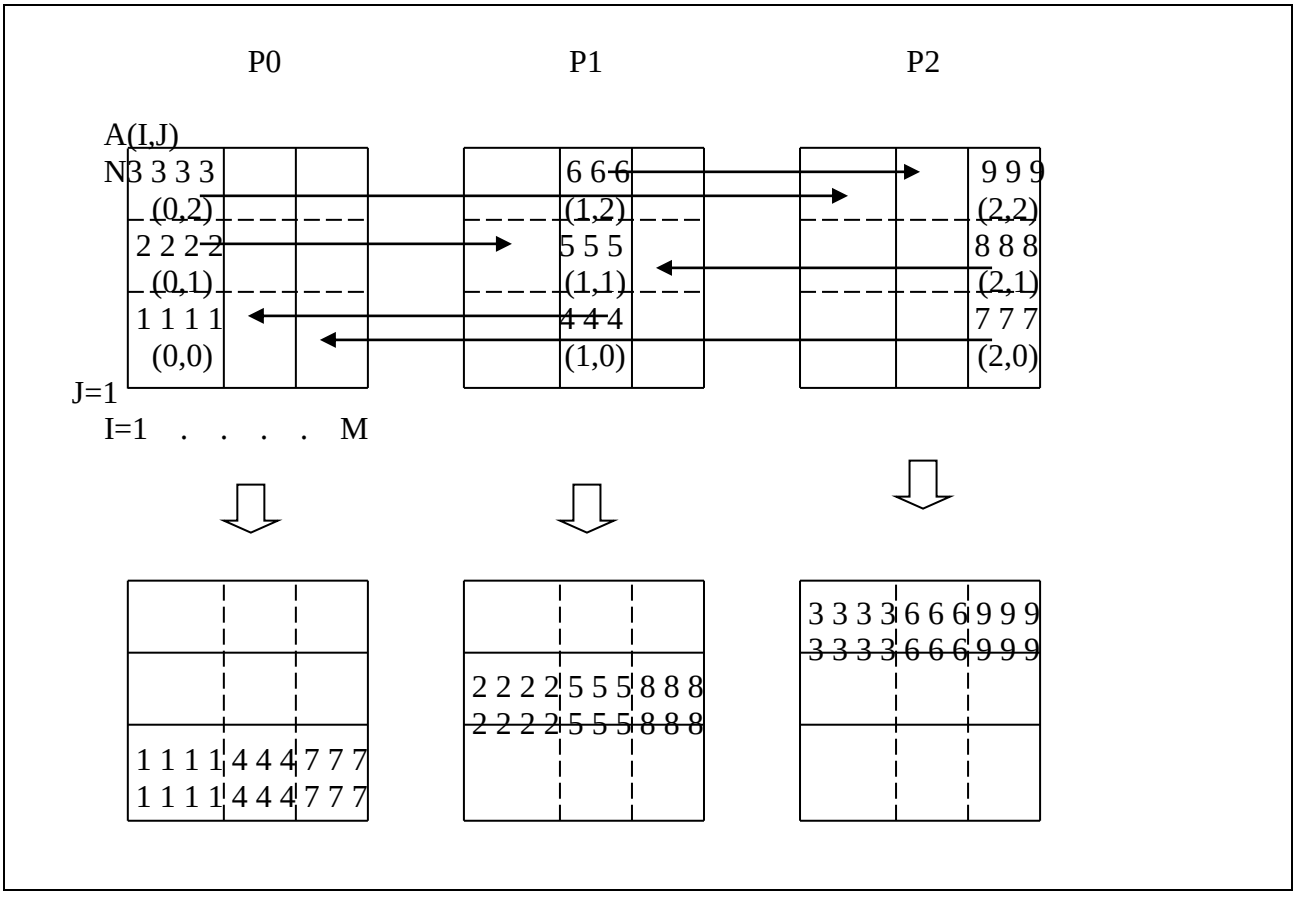


圖 7.5　col_to_row　transpose

使用 BLOCK2D 資料類別　col_to_row　之程式片段如下

```
 ITAG=20
 DO ID=0, NPROC-1
   IF (MYID.NE.ID) THEN
     ISTART1=ISTARTG(ID)
     JSTART1=JSTARTG(ID)
     CALL MPI_ISEND( A(ISTART,JSTART1), 1, ITYPE(MYID,ID), ID, ITAG,
1                          MPI_COMM_WORLD, IREQ1(ID), IERR)
     CALL MPI_IRECV( A(ISTART1,JSTART), 1, ITYPE(ID,MYID), ID, ITAG,
2                          MPI_COMM_WORLD, IREQ2(ID), IERR)
   ENDIF
 ENDDO
 DO ID=0, NPROC-1
   IF(ID.NE.MYID) THEN
     CALL MPI_WAIT(IREQ1(ID), ISTATUS,IERR)
     CALL MPI_WAIT(IREQ2(ID), ISTATUS,IERR)
   ENDIF
 ENDDO
```

使用 BLOCK2D 料類別之測試程式如下

```
 SUBROUTINE BLOCK2D (IMIN, IMAX, ILENG, JLENG, OLDTYPE,NEWTYPE)
 INCLUDE       'mpif.h'
 INTEGER       IMIN, IMAX, ILENG, JLENG, OLDTYPE, NEWTYPE
 ISTRIDE = IMAX - IMIN +1
 CALL MPI_TYPE_VECTOR (JLENG,ILENG,ISTRIDE, OLDTYPE, NEWTYPE,IERR)
 CALL MPI_TYPE_COMMIT ( NEWTYPE, IERR)
 RETURN
 END

 PROGRAM TRANSPOSE
 INCLUDE    'mpif.h'
 PARAMETER (NP=3, M=9,N=6)
```

```
      REAL*8       A(M,N)
      INTEGER    ITYPE(0:NP, 0:NP), IREQ1(0:NP), IREQ2(0:NP),
     &           ISTARTG(0:NP), IENDG(0:NP), JSTARTG(0:NP), JENDG(0:NP)
      INTEGER    ISTATUS(MPI_STATUS_SIZE), ISTAT(MPI_STATUS_SIZE, NP)
      CALL MPI_INIT( IERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYID, IERR)
      CALL MPI_BARRIER( MPI_COMM_WORLD, IERR)
      DO J=0, NPROC-1
         CALL STARTEND( J, NPROC, 1, N, JSTARTG(J),JENDG(J) )
      ENDDO
      DO I=0,NPROC-1
         CALL STARTEND(I, NPROC, 1, M, ISTARTG(I), IENDG(I) )
      ENDDO
      DO J=0, NPROC-1
        JLENG=JENDG(J)-JSTARTG(J)+1
        DO I=0, NPROC-1
          ILENG=IENDG(I)-ISTARTG(I)+1
          IEND=IENDG(I)
          CALL TYPE_BLOCK2A (1, M, ILENG, JLENG, MPI_REAL8, ITEMP)
          ITYPE(I,J)=ITEMP
        ENDDO
      ENDDO
      ISTART=ISTARTG(MYID)
      IEND=IENDG(MYID)
      JSTART=JSTARTG(MYID)
      JEND=JENDG(MYID)
101   FORMAT (9F5.0)
      AID=MYID
      DO J=JSTART,JEND
        DO I=1,3
          A(I,J)=1.0+AID
        ENDDO
        DO I=4,6
          A(I,J)=4.0+AID
        ENDDO
        DO I=7,9
          A(I,J)=7.0+AID
```

```
          ENDDO
        ENDDO
        IU=MYID+11
        DO J=JSTART,JEND
          WRITE(IU,101) (A(I,J),I=1,M)
        ENDDO
C
C       row_to_col
C
        ITAG=10
        DO ID=0, NPROC-1
          IF (ID.NE.MYID) THEN
            ISTART1=ISTARTG(ID)
            JSTART1=JSTARTG(ID)
            CALL MPI_ISEND( A(ISTART1,JSTART), 1, ITYPE(ID, MYID), ID, ITAG,
     1                      MPI_COMM_WORLD, IREQ1(ID), IERR)
            CALL MPI_IRECV( A(ISTART,JSTART1), 1, ITYPE(MYID, ID), ID, ITAG,
     2                      MPI_COMM_WORLD, IREQ2(ID), IERR)
          ENDIF
        ENDDO
        DO ID=0, NPROC-1
          IF(ID.NE.MYID) THEN
            CALL MPI_WAIT( IREQ1(ID), ISTATUS,IERR)
            CALL MPI_WAIT( IREQ2(ID), ISTATUS,IERR)
          ENDIF
        ENDDO
        WRITE(IU,*) 'after row_to_col'
        DO J=1,N
          WRITE(IU,101) (A(I,J),I=ISTART,IEND)
        ENDDO
        DO J=1,N
          DO I=ISTART,IEND
            A(I,J)=A(I,J)+10.0
          ENDDO
        ENDDO
C
C       col_to_row
C
```

```
  ITAG=20
  DO ID=0, NPROC-1
     IF (ID.NE.MYID) THEN
        ISTART1=ISTARTG(ID)
        JSTART1=JSTARTG(ID)
        CALL MPI_ISEND( A(ISTART,JSTART1), 1, ITYPE(MYID,ID), ID, ITAG,
1                       MPI_COMM_WORLD, IREQ1(ID), IERR)
        CALL MPI_IRECV( A(ISTART1,JSTART), 1, ITYPE(ID,MYID), ID, ITAG,
2                       MPI_COMM_WORLD, IREQ2(ID), IERR)
     ENDIF
  ENDDO
  DO ID=0, NPROC-1
     IF(ID.NE.MYID) THEN
        CALL MPI_WAIT (IREQ1(ID), ISTATUS, IERR)
        CALL MPI_WAIT (IREQ2(ID), ISTATUS, IERR)
     ENDIF
  ENDDO
WRITE(IU,*) 'after col_to_row'
DO J=JSTART,JEND
     WRITE(IU,101) (A(I,J),I=1,M)
ENDDO
CALL MPI_FINALIZE(IERR)
STOP
END
```

上述 TRANSPOSE 程式在三個 CPU 上執行的結果，fort.11 檔案之內容如下。

```
   1.    1.    1.    4.    4.    4.    7.    7.    7.
   1.    1.    1.    4.    4.    4.    7.    7.    7.
after row_to_col
   1.    1.    1.
   1.    1.    1.
   2.    2.    2.
   2.    2.    2.
   3.    3.    3.
   3.    3.    3.
after col_to_row
```

```
 11.  11.  11.  14.  14.  14.  17.  17.  17.
 11.  11.  11.  14.  14.  14.  17.  17.  17.
```

fort.12 案之內容如下。

```
  2.   2.   2.   5.   5.   5.   8.   8.   8.
  2.   2.   2.   5.   5.   5.   8.   8.   8.
 after row_to_col
  4.   4.   4.
  4.   4.   4.
  5.   5.   5.
  5.   5.   5.
  6.   6.   6.
  6.   6.   6.
 after col_to_row
 12.  12.  12.  15.  15.  15.  18.  18.  18.
 12.  12.  12.  15.  15.  15.  18.  18.  18.
```

fort.13 之內容如下。

```
  3.   3.   3.   6.   6.   6.   9.   9.   9.
  3.   3.   3.   6.   6.   6.   9.   9.   9.
 after row_to_col
  7.   7.   7.
  7.   7.   7.
  8.   8.   8.
  8.   8.   8.
  9.   9.   9.
  9.   9.   9.
 after col_to_row
 13.  13.  13.  16.  16.  16.  19.  19.  19.
 13.  13.  13.  16.  16.  16.  19.  19.  19.
```

在 col_to_row 或 row_to_col 轉換時，各別 MPI_ISEND、MPI_IRECV 之後的等待指令

MPI_WAIT 也可以用 MPI_WAITALL 來取代它，其叫用格式如下 ：

CALL MPI_WAITALL (COUNT, REQUEST, STATUS, IERR)

其中之引數

COUNT	MPI_ISEND 或 MPI_IRECV 之次數，整常數或整變數
REGUEST	每個傳送之編號，COUNT 個元素的整數陣列
STATUS	每個等待的結果，COUNT 個元素的整數陣列

由於在 row_to_col 或 col_to_row 的轉換程式段落裏，當 ID 等於 MYID 時 IREQ1(ID)及 IREQ2(ID)的值沒有設定，不能直接用做 MPI_WAITALL 的引數。可將轉換程式段落裏稍做修改，使得 REQUEST 陣列裏備有正確的數值即可。例如 row_to_col 的轉換程式段落作如下的修改之後，就可以使用 MPI_WAITALL。

```
PARAMETER (NP=16)
INTEGER      IREQ1(NP), IREQ2(NP), ISTAT(MPI_STATUS_SIZE, NP)
.  .  .  .  .  .  .
ITAG=10
K=0
DO ID=0, NPROC-1
    IF (ID.NE.MYID) THEN
      K=K+1
      ISTART1=ISTARTG(ID)
      JSTART1=JSTARTG(ID)
      CALL MPI_ISEND( A(ISTART1,JSTART), 1, ITYPE(ID, MYID), ID, ITAG,
1                          MPI_COMM_WORLD, IREQ1(K), IERR)
      CALL MPI_IRECV( A(ISTART,JSTART1), 1, ITYPE(MYID, ID), ID, ITAG,
2                          MPI_COMM_WORLD, IREQ2(K), IERR)
    ENDIF
ENDDO
```

```
KOUNT=NPROC-1
CALL    MPI_WAITALL (KOUNT, IREQ1, ISTAT, IERR)
CALL    MPI_WAITALL (KOUNT, IREQ2, ISTAT, IERR)
```

此處的 ISTAT 必須設定為

```
INTEGER    ISTAT( MPI_STATUS_SIZE,    NP)
```

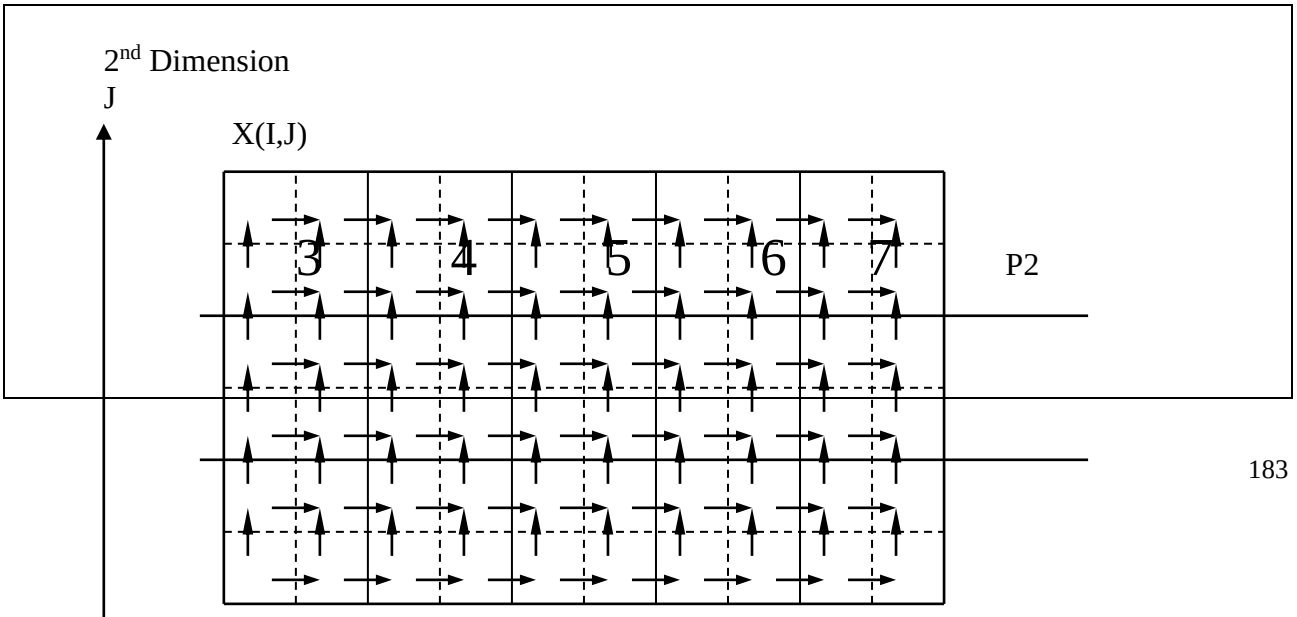
採用 MPI_WAIT 或 MPI_WAITALL 的執行效率沒有明顯的差異。

7.3 兩方迴歸與管線法

前面所探討的程式在其 DO loop 裏的陳述，等號左邊的變數和等號右邊的變數是不同的。現在開始考慮兩者相同的情況。例如下面這一個 DO loop 裏等號兩邊的變數都是 X，等號左邊的 index 是(I,J)而等號右邊的 index 則有 I-1 和 J-1，I 和 J 兩個方向都是迴歸 (Recursive)，是為兩方迴歸 (2-Way Recursive) 。這一種程式的平行化比較麻煩，首先介紹管線法 (Pipeline Method)，其他方法請參閱第八章。

```
PARAMETER    (MX=128, MY=128)
REAL              X(0:MX, 0:MY)
.  .  .  .  .  .  .  .
DO J=1,MY
DO I=1,MX
   X(I,J)=X(I,J)+( X(I-1,J)+X(I,J-1) )*0.5
ENDDO
ENDDO
```

X 陣列是在第二維上做區段切割，如圖 7.2。圖裏的箭頭代表資料的依賴關係，水平方向第一維的依賴關係存在於各個 CPU 的內部，不會構成問題，但是垂直方向第二維的依賴關係則有跨越 CPU 邊界的情形。只有在 I 方向每計算一小段之後把邊界資料傳送給下一個 CPU，然後



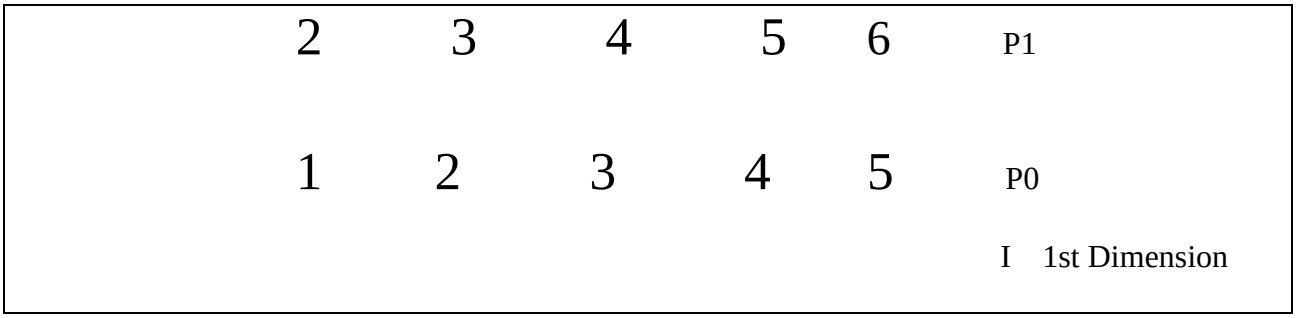


圖 7.2 (a) 管線法的資料切割與區塊切割

再繼續計算下一個小段。後面的 CPU 要等到收到前一個 CPU 送來的邊界資料之後，才能開始該小段的計算工作。所以，後一個 CPU 永遠比它前一個 CPU 晚一步，如圖 7.2(b)所示。I 方向切得越細則可以平行的比例越高，但是資料交換的次數也越多，耗費在資料交換的時間也越長。I 方向切得越粗則可以平行的比例越低，平行的效果也越差。

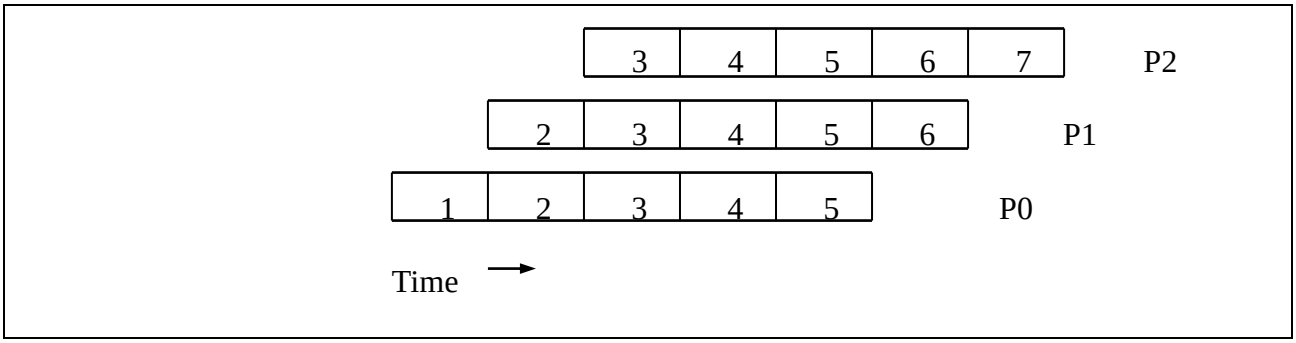


圖 7.2 (b) 管線法的區塊執行順序圖

管線法的循序程式如下

```
      PROGRAM PIPESEQ
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (M=128,N=128)
      REAL*8    X(0:M+1,0:N+1)
C
      CALL SYSTEM_CLOCK(IC,IR,IM)
```

```
      CLOCK=DBLE(IC)/DBLE(IR)
      OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
      READ(7)    X
      EPS=1.0D-5
      OMEGA=0.5
      DO 900 LOOP=1,6000
        ERR1=0.0
        DO J=1,N
        DO I=1,M
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        IF(ERR1.LT.EPS) GOTO 901
 900  CONTINUE
 901  CONTINUE
      WRITE(*,*) 'LOOP,ERR1=', LOOP,ERR1
      WRITE(*,*) 'X(M,J),J=1,', N, ',8'
      WRITE(*,101) (X(M,J),J=1,N,8)
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE(IC)/DBLE(IR) - CLOCK
      PRINT *,' ELAPSED TIME=', CLOCK
 101  FORMAT(8E10.3)
      STOP
      END
```

管線法的循序程式 PIPESEQ 在 IBM SP2 SMP 上執行的結果如下

```
LOOP,ERR1= 4875    0.999757520445815340E-05
 X(M,J),J=1, 128 ,8
  .382E+00   .116E+00   .661E-01   .477E-01   .382E-01   .324E-01   .284E-01   .256E-01
  .234E-01   .217E-01   .204E-01   .192E-01   .183E-01   .175E-01   .167E-01   .161E-01
 ELAPSED TIME= 5.99000000000523869
```

管線法的平行程式如下

```
      PROGRAM PIPELINE
      IMPLICIT REAL*8 (A-H,O-Z)
C                                        Parallel on 2nd dimension
      INCLUDE     'mpif.h'
      PARAMETER (M=128,N=128,NLOOP=10)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,EPSILON,ABS
C-----------------------------------------------------------------
      INTEGER    NPROC,MYID,ISTATUS(MPI_STATUS_SIZE),JSTART,JEND,
     1           JSTARTM1,JENDP1,L_NBR,R_NBR
      COMMON/COMM/
     1           NPROC,MYID,ISTATUS,JSTART,JEND,L_NBR,R_NBR,LASTP,
     2           JSTARTG(0:31),JENDG(0:31)
C-----------------------------------------------------------------
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK ( MPI_COMM_WORLD, MYID, IERR)
      CALL MPI_BARRIER     ( MPI_COMM_WORLD, IERR)
      CLOCK=MPI_WTIME()
      LASTP=NPROC-1
      IF(MYID.EQ.0) THEN
        OPEN(7,FILE='input.dat',FORM='unformatted')
        READ(7) X
      ENDIF
      KOUNT=(M+2)*(N+2)
      CALL MPI_BCAST(X, KOUNT, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
      DO I=0,NPROC-1
        CALL STARTEND( I, NPROC, 1, N, JSTARTG(I), JENDG(I))
      ENDDO
      JSTART=JSTARTG(MYID)
      JEND=JENDG(MYID)
      JSTARTM1=JSTART-1
      JENDP1=JEND+1
      PRINT *,' MYID,JSTART,JEND=', MYID,JSTART,JEND

      L_NBR=MYID-1
      R_NBR=MYID+1
      IF(MYID.EQ.0)              L_NBR=MPI_PROC_NULL
```

```
      IF(MYID.EQ.NPROC-1) R_NBR=MPI_PROC_NULL
C
      IBLOCK=4
      OMEGA=0.5
      EPS=1.0D-5
      DO 998 LOOP=1,8000
        ERR1=1.0D-15
        ITAG=20
        CALL MPI_SENDRECV (X(1,JSTART), M, MPI_REAL8, L_NBR, ITAG,
     1                     X(1,JENDP1), M, MPI_REAL8, R_NBR, ITAG,
     2                     MPI_COMM_WORLD, ISTATUS, MPI_ERR)
        ITAG=10
        DO II=1,M,IBLOCK
          IBLKLEN=MIN(IBLOCK, M-II+1)
          CALL MPI_RECV( X(II,JSTARTM1),IBLKLEN,MPI_REAL8,L_NBR,ITAG,
     1                   MPI_COMM_WORLD,ISTATUS,IERR)
          DO J=JSTART,JEND
          DO I=II,II+IBLKLEN-1
            TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
            X(I,J)=X(I,J)+OMEGA*TEMP
            ABST=DABS(TEMP)
            IF(ABST.GT.ERR1) ERR1=ABST
          ENDDO
          ENDDO
          CALL MPI_SEND( X(II,JEND), IBLKLEN, MPI_REAL8, R_NBR, ITAG,
     1                   MPI_COMM_WORLD, IERR)
        ENDDO
        CALL MPI_ALLREDUCE( ERR1, GERR1, 1, MPI_REAL8, MPI_MAX,
     1                      MPI_COMM_WORLD, MPI_ERR)
        ERR1=GERR1
        IF(ERR1.LT.EPS) GOTO 999
  998 CONTINUE
  999 CONTINUE
      IF( MYID.EQ.0) THEN
        DO J=1,LASTP
          JSTART1=JSTARTG(J)
          KOUNT=(JENDG(J)-JSTART1+1)*(M+2)
          CALL MPI_RECV (X(0,JSTART1), KOUNT, MPI_REAL8, J, ITAG,
```

```
     1                          MPI_COMM_WORLD, ISTATUS, IERR)
        ENDDO
        WRITE(*,*) 'LOOP,ERR1=', LOOP,ERR1
        WRITE(*,*) 'X(M,J),J=1,',N,',8'
        WRITE(*,101) ( X(M,J),J=1,N,8)
      ELSE
        KOUNT=(JEND-JSTART+1)*(M+2)
        CALL MPI_SEND (X(0,JSTART), KOUNT, MPI_REAL8, 0, ITAG,
     1                      MPI_COMM_WORLD, IERR)
      ENDIF
      CLOCK=MPI_WTIME() – CLOCK
      PRINT *,' MYID, ELAPSED TIME=', MYID, CLOCK
101   FORMAT(8E10.3)
      CALL MPI_FINALIZE( IERR)
      STOP
      END
```

管線法的平行程式 PIPELINE 在 IBM SP2 SMP 的四棵 CPU 上執行的結果如下，耗時 6.16 秒，

比循序程式的 5.99 秒還慢。較有效率的平行方法請參考第八章。

、，。

```
ATTENTION: 0031-408    4 tasks allocated by LoadLeveler, continuing...
   MYID,JSTART,JEND= 0      1    32
   MYID,JSTART,JEND= 1    33    64
   MYID,JSTART,JEND= 2    65    96
   MYID,JSTART,JEND= 3    97    128
   MYID, ELAPSED TIME= 1    6.15884946845471859
   MYID, ELAPSED TIME= 2    6.15885968133807182
   MYID, ELAPSED TIME= 3    6.15914255287498236
  LOOP,ERR1= 4875    0.99975752044581534OE-05
  X(M,J),J=1, 128 ,8
    .382E+00    .116E+00    .661E-01    .477E-01    .382E-01    .324E-01    .284E-01    .256E-01
    .234E-01    .217E-01    .204E-01    .192E-01    .183E-01    .175E-01    .167E-01    .161E-01
    MYID, ELAPSED TIME= 0 6.15982436109334230
```

第八章　多方依賴及 SOR 解法

本章將討論一個格點的上下左右四個方向或八個方向都有依賴關係的 SOR(Successive Over-Relaxation)解法。

第一節 介紹 SOR 解法。

第二節 介紹黑白點間隔 SOR 解法。

第三節 介紹斑馬線 SOR 解法。

第四節 介紹四色點間隔 SOR 解法。

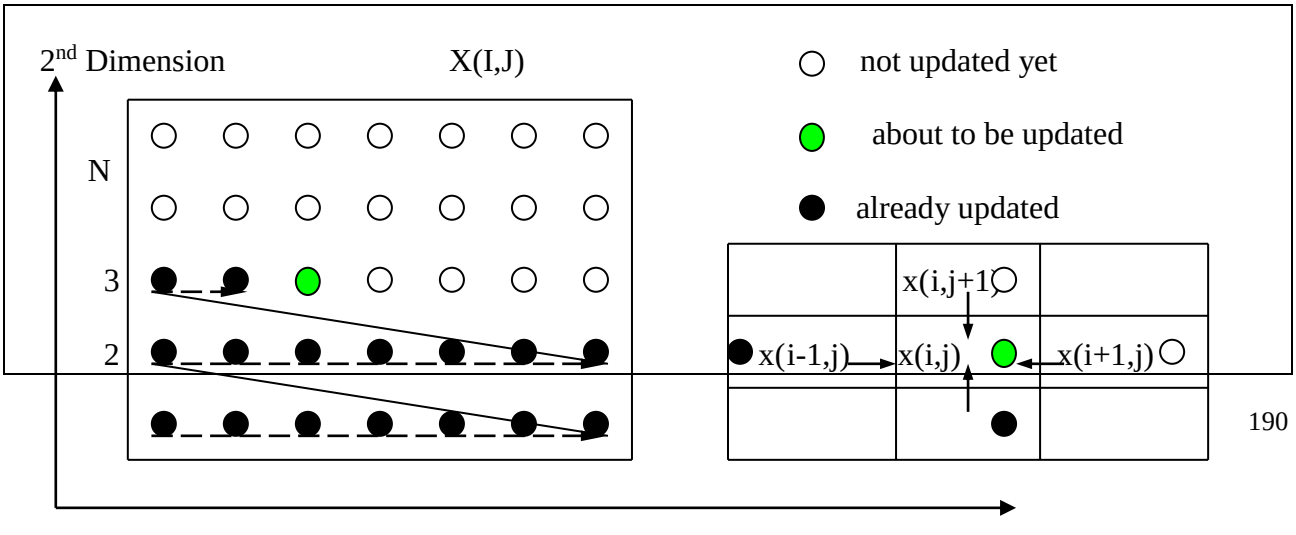
8.1 四方依賴及 SOR 解法

下列程式是使用 Successive Over-Relaxation (SOR) method 解二維的 Laplace 方程式。最外層的 DO-loop 是要使 x 達到收斂， 而裡層的 DO-loop 是用來更新 x 的值。收斂的速度可用控制參數 omega 來加以改變。

該裡層的 DO-loop 如下

```
DO J=1,N
DO I=1,M
   TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
   X(I,J)=X(I,J)+OMEGA*TEMP
   ABST=ABS(TEMP)
   IF(ABST.GT.ERR1) ERR1=ABST
ENDDO
ENDDO
```

陣列元素 x(i,j) 的值係由其左、右、上、下四個元素的值計算得來的，如圖 8.1 所示。其左邊的 x(i-1,j)和下面的 x(i,j-1)之值是已經更新過的，而其右邊的 x(i+1,j)和上面的 x(i,j+1)之值則是尚未更新過的。要將這一種程式加以平行，除了管線法 (pipeline method) 之外，還可以採用黑白點間隔法 (red-black SOR method) 。管線法效率較差，最好不用它。



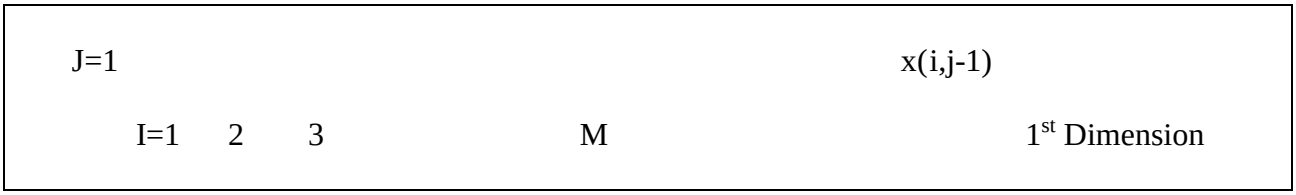


圖 8.1　SOR 循序計算示意圖

```
      PROGRAM    SORSEQ
C
C       Sequential Version of SOR method
C
      PARAMETER (M=128, N=128, NLOOP=6000)
      REAL*8    X(0:M+1,0:N+1), TEMP, OMEGA, ERR1, EPSILON, ABST, CLOCK
C
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE(IC)/ DBLE (IR)
      OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
      READ(7)    X
      EPSILON=1.0D-5
      OMEGA=0.5
      DO 900 ITER=1, NLOOP
        ERR1=0.0
        DO J=1,N
        DO I=1,M
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        IF(ERR1.LE.EPSILON) GOTO 901
 900  CONTINUE
 901  CONTINUE
       WRITE(*,*) 'LOOP, ERR1=', LOOP, ERR1
       WRITE(*,*) 'X(M,J),J=1,' N, ',8'
       WRITE(*,101) (X(M,J),J=1,N,8)
       CALL SYSTEM_CLOCK(IC,IR,IM)
```

```
      CLOCK= DBLE (IC)/ DBLE (IR) - CLOCK
      PRINT *,' ELAPSED TIME=', CLOCK
101   FORMAT(8E10.3)
      STOP
      END
```

其輸入資料檔案 input.dat 係由下列程式所產生。

```
      PROGRAM    SOR_DATA
      PARAMETER (M=128, N=128)
      REAL*8    X(0:M+1,0:N+1)
C
      DO J=0,N+1
        X(0,J)=0.0
        X(J,0)=0.0
      ENDDO
      DO J=1,N+1
        AJ=1.D0/DBLE(J)
        DO I=1,M+1
          X(I,J)=AJ+1.D0/DBLE(I)
        ENDDO
      ENDDO
      OPEN(7,FILE='input.dat',FORM='UNFORMATTED')
      WRITE(7) X
      CLOSE(7)
      STOP
      END
```

SORSEQ 循序程式在 IBM SP2 SMP 執行的結果如下。當 ITER 到達 4875 次時其最大誤差值已經小於 EPSILON 而達到收斂的狀態，共耗時 5.99 秒。

```
 LOOP, ERR1= 4875      0.9997575204458153 40E-05
 X(M,J),J=1,    128    ,8
  .382E+00    .116E+00    .661E-01    .477E-01    .382E-01    .324E-01    .284E-01    .256E-01
```

.234E-01 .217E-01 .204E-01 .192E-01 .183E-01 .175E-01 .167E-01 .161E-01
ELAPSED TIME= 5.992187500

8.2 黑白點間隔 SOR 解法

黑白點間隔法 (red-black SOR method) 是先計算所有的白色格點，然後計算所有的黑色格點，如圖 8.2。也就是當

i+j 的值是偶數時為白色格點

i+j 的值是奇數時為黑色格點

計算黑色格點時其四周都是白色格點，而計算白色格點其四周都是黑色格點。因此，每一個黑色格點都可以單獨計算，白色格點也是如此。這一種計算方法是從原來的 SOR 演算法修改過來的，可能造成收斂速度的改變，以及最後各個格點數值的改變，但是卻較容易平行化。

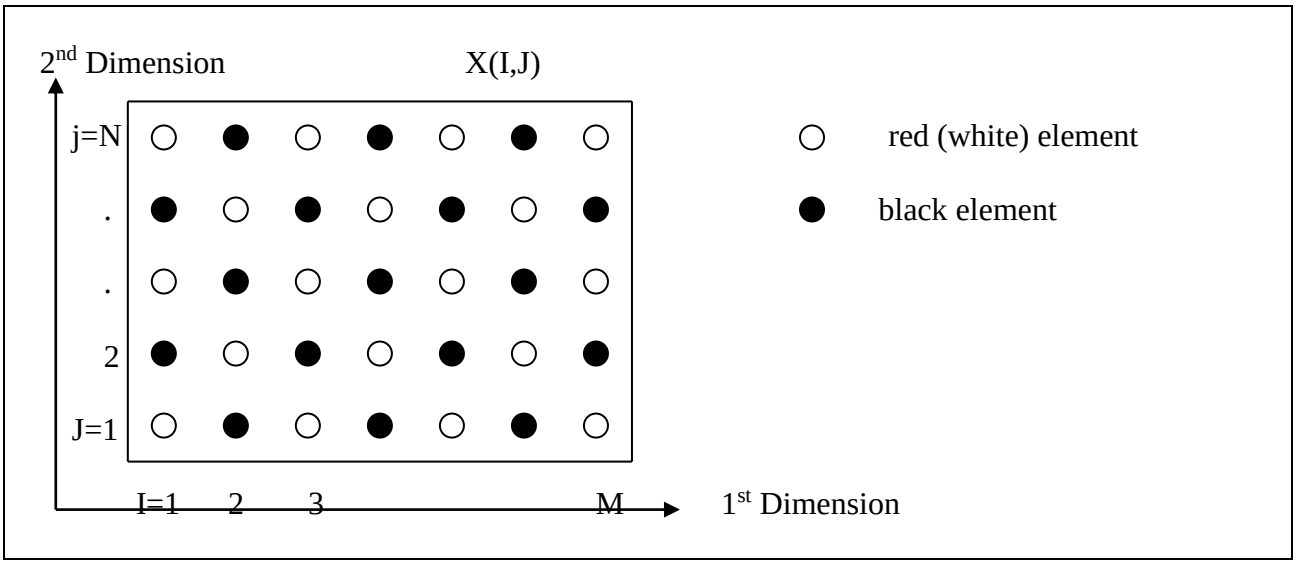


圖 8.2 黑白點間隔法 SOR 解法示意圖

黑白點間隔法的循序程式片段如下。首先計算所有的白色格點，也就是 j+i 為偶數的格點。然後才計算黑色格點，也就是 j+i 為奇數的格點。

```
      PROGRAM SOR_RB
C
C      Sequential version of red-black SOR (Successive Over-Relaxation Method)
C
      PARAMETER (M=128, N=128)
      REAL*8   X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,EPSILON,ABST, CLOCK
C
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE(IC)/ DBLE (IR)
      OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
      READ(7)   X
      EPSILON=1.0D-5
      OMEGA=0.5
      DO 900 ITER=1,6000
        ERR1=0.0

        DO J=1,N                    ! 白色格點

        DO I=MOD(J+1,2)+1,M,2
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO

        DO J=1,N                    ! 黑色格點

        DO I=MOD(J,2)+1,M,2
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        IF(ERR1.LE.EPSILON) GOTO 910
  900 CONTINUE
  910 CONTINUE
      WRITE(*,*) 'ITER, ERR1=', ITER, ERR1
      WRITE(*,*) 'X(M,J),J=1,', N, ',8'
```

```
      WRITE(*,101) (X(M,J),J=1,N,8)
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK= DBLE (IC)/ DBLE (IR) - CLOCK
      PRINT *,' ELAPSED TIME=', CLOCK
101   FORMAT(8E10.3)
      STOP
      END
```

循序程式 SOR**_RB** 在 IBM SP2 SMP 上執行的結果如下。當 ITER 到達 4929 次時其最大誤差值已經小於 EPSILON 而達到收斂的狀態，共耗時 2.32 秒。比原來 SOR 演算法的 4875 次稍多些，但是僅耗時 2.32 秒，比 SOR 的 5.99 秒快了一倍多。

```
 ITER, ERR1= 4929    0.99979296301483389 1E-05
  X(M,J),J=1, 128 ,8
    .382E+00   .116E+00   .661E-01   .477E-01   .382E-01   .324E-01   .284E-01   .256E-01
    .234E-01   .217E-01   .204E-01   .192E-01   .183E-01   .175E-01   .167E-01   .161E-01
 ELAPSED TIME= 2.31999999999970896
```

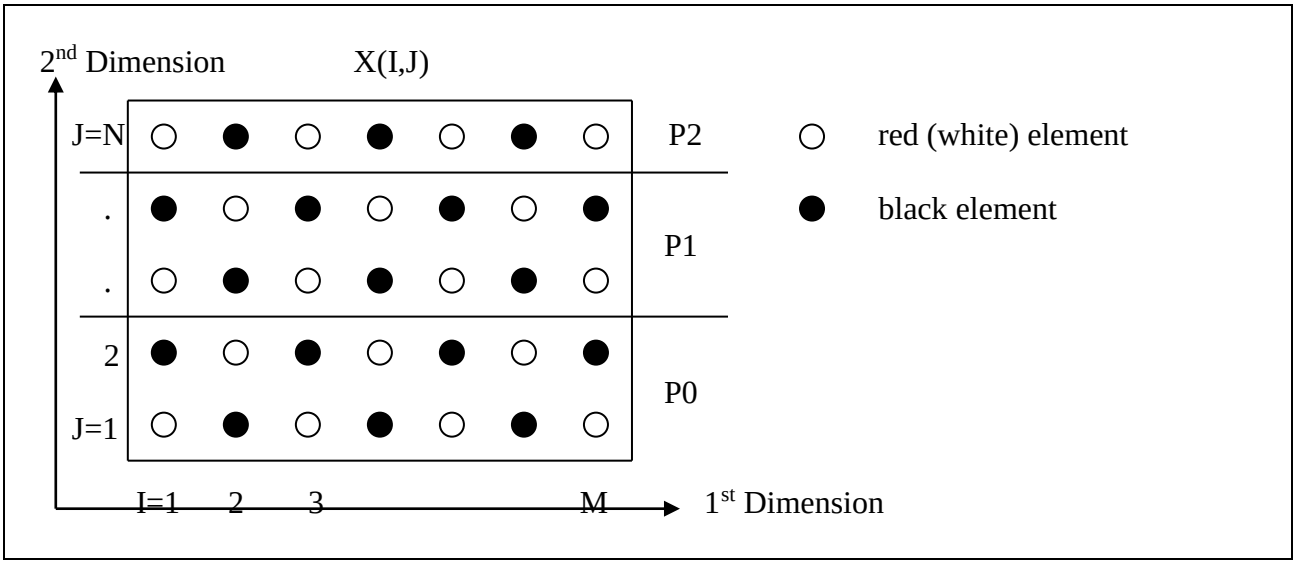


圖 8.3 在第二維做平行計算的 red_black SOR method

在第二維做平行計算的 Red-Black SOR 程式如下。在計算白色格點之前，叫用 COMMU(0) 取得界外的黑色格點資料，在計算過白色格點之後要計算黑色格點之前，叫用 COMMU(1) 取得界外的白色格點資料。

```
      PROGRAM SOR_RBP
C
C      Parallel version of red-black SOR Method - Parallel along the 2nd dimension
C
      INCLUDE     'mpif.h'
      PARAMETER (M=128,N=128)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,GERR1,EPS,ABST,CLOCK
      INTEGER ISTATUS(MPI_STATUS_SIZE),JSTART,JEND,
     1              JSTARTM1, JENDP1, L_NBR, R_NBR
      REAL*8    BUFR1(M+2), BUFS1(M+2), BUFR2(M+2), BUFS2(M+2)
      COMMON/COMM/BUFR1, BUFS1, BUFR2, BUFS2,
     1              NPROC, MYID, ISTATUS, JSTART, JEND, L_NBR, R_NBR, LASTP,
     2              JSTARTG(0:31), JENDG(0:31)
C
      CALL MPI_INIT (MPI_ERR)
      CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NPROC, MPI_ERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, MPI_ERR)
      CLOCK=MPI_WTIME()
      LASTP=NPROC-1
      DO I=0, LASTP
         CALL STARTEND (I, NPROC, 1, N, JSTART, JEND)
         JSTARTG(I)=JSTART
         JENDG(I)=JEND
      ENDDO
      JSTART=JSTARTG(MYID)
      JEND=JENDG(MYID)
      PRINT *,' NPROC, MYID, JSTART, JEND=', NPROC, MYID, JSTART, JEND
      JSTARTM1=JSTART-1
      JENDP1=JEND+1

      L_NBR=MYID-1
```

```
 R_NBR=MYID+1
 IF(MYID.EQ.0)          L_NBR=MPI_PROC_NULL
 IF(MYID.EQ.LASTP) R_NBR=MPI_PROC_NULL

 IF (MYID.EQ.0) THEN
    OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
    READ(7)    X
 ENDIF
 KOUNT=(M+2)*(N+2)
 CALL MPI_BCAST (X, KOUNT, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
 EPS=1.0D-5
 OMEGA=0.5
 DO 900    ITER=1,6000
    ERR1=0.0
    CALL COMMU(0)
    DO J=JSTART,JEND

    DO I=MOD(J+1,2)+1,M,2                ! 白色格點

       TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
       X(I,J)=X(I,J)+OMEGA*TEMP
       ABST=ABS(TEMP)
       IF(ABST.GT.ERR1) ERR1=ABST
    ENDDO
    ENDDO
    CALL COMMU(1)

    DO J=JSTART, JEND                ! 黑色格點

    DO I=MOD(J,2)+1,M,2
       TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
       X(I,J)=X(I,J)+OMEGA*TEMP
       ABST=ABS(TEMP)
       IF(ABST.GT.ERR1) ERR1=ABST
    ENDDO
    ENDDO
    CALL MPI_ALLREDUCE( ERR1, GERR1, 1, MPI_REAL8, MPI_MAX,
1                                   MPI_COMM_WORLD, IERR)
    ERR1=GERR1
    IF(ERR1 .LE. EPS) GOTO 910
```

```
 900  CONTINUE
 910  CONTINUE
      IF( MYID.EQ.0) THEN
        DO I=1,LASTP
          JSTART1=JSTARTG(I)
          KOUNT=(JENDG(I)-JSTART1+1)*(M+2)
          CALL MPI_RECV (X(0,JSTART1), KOUNT, MPI_REAL8, I, ITAG,
     1                                MPI_COMM_WORLD, ISTATUS, IERR)
        ENDDO
        WRITE(*,*) 'ITER, ERR1=', ITER, ERR1
        WRITE(*,*) 'X(M,J),J=1', N,',8'
        WRITE(*,101) (X(M,J),J=1,N,8)
      ELSE
        KOUNT=(JEND-JSTART+1)*(M+2)
        CALL MPI_SEND (X(0,JSTART), KOUNT, MPI_REAL8, 0, ITAG,
     1                             MPI_COMM_WORLD, IERR)
      ENDIF
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,' MYID, ELAPSED TIME=', MYID, CLOCK
 101  FORMAT(8E10.3)
      CALL MPI_FINALIZE( MPI_ERR)
      STOP
      END
      SUBROUTINE COMMU(IFLAG)
      INCLUDE    'mpif.h'
      INTEGER    IFLAG
C
C       IFLAG=0    for black elements
C       IFLAG=1    for red    elements
C
      PARAMETER (M=128,N=128)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,GERR1,EPSILON,ABST,CLOCK
C - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      INTEGER    NPROC,MYID,STATUS(MPI_STATUS_SIZE),JSTART,JEND,
     1           JSTARTM1,JENDP1,LEFT_NBR,RIGHT_NBR
      REAL*8     BUFR1(M+2), BUFS1(M+2), BUFR2(M+2), BUFS2(M+2)
      COMMON/COMM/BUFR1,BUFS1,BUFR2,BUFS2,
     1           NPROC,MYID,STATUS,JSTART,JEND,L_NBR,R_NBR,LASTP
```

```
C - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      IS1=MOD(JSTART+IFLAG, 2) +1
      IS2=MOD(JEND+IFLAG, 2) +1
      IR1 = 3 - IS1
      IR2 = 3 - IS2
      IF(MYID.NE.0) THEN
        ICNT1=0
        DO I=IS1, M, 2
          ICNT1 = ICNT1 + 1
          BUFS1(ICNT1)=X(I,JSTART)
        ENDDO
      ENDIF
      IF (MYID.NE.LASTP) THEN
        ICNT2=0
        DO I=IS2, M, 2
          ICNT2 = ICNT2 + 1
          BUFS2(ICNT2) = X(I,JEND)
        ENDDO
      ENDIF
      CALL MPI_SENDRECV (BUFS1, ICNT1, MPI_REAL8,    L_NBR, 1,
     1                   BUFR2, M+2    MPI_REAL8,    R_NBR, 1,
     2                   MPI_COMM_WORLD, STATUS, IERR)
      CALL MPI_SENDRECV (BUFS2, ICNT2, MPI_REAL8, R_NBR, 2,
     1                   BUFR1, M+2,    MPI_REAL8, L_NBR, 2,
     2                   MPI_COMM_WORLD, STATUS, IERR)
      IF(MYID.NE.0) THEN
        ICNT=0
        DO I=IR1, M, 2
          ICNT = ICNT + 1
          X(I,JSTARTM1) = BUFR1(ICNT)
        ENDDO
      ENDIF
      IF(MYID.NE.LASTP) THEN
        ICNT=0
        DO I=IR2, M, 2
          ICNT = ICNT + 1
          X(I,JENDP1) = BUFR2(ICNT)
        ENDDO
```

```
      ENDIF
      END
      SUBROUTINE STARTEND(MYID,NPROC,IS1,IS2,ISTART,IEND)
      INTEGER MYID,NPROC,IS1,IS2,ISTART,IEND
      ILENGTH=IS2-IS1+1
      IBLOCK=ILENGTH/NPROC
      IR=ILENGTH-IBLOCK*NPROC
      IF(MYID.LT.IR) THEN
        ISTART=IS1+MYID*(IBLOCK+1)
        IEND=ISTART+IBLOCK
      ELSE
        IEND=ISTART+IBLOCK-1
      ENDIF
      IF(ILENGTH.LT.1) THEN
        ISTART=1
        IEND=0
      ENDIF
      RETURN
      END
```

平行版 SOR**_RBP** 程式在 IBM SP2 SMP 四棵 CPU 上執行的結果如下。當 ITER 到達 2293 次時其最大誤差值已經小於 EPSILON 而達到收斂的狀態，共耗時 1.36 秒。比循序版 Red-Black SOR 演算法的 4929 次少了一半以上，收斂得快了很多。平行績效(speed up) =2.33/1.36=1.71 倍。

```
ATTENTION: 0031-408    4 tasks allocated by LoadLeveler, continuing...
   NPROC,MYID,JSTART,JEND= 4    1    33    64
   NPROC,MYID,JSTART,JEND= 4    2    65    96
   NPROC,MYID,JSTART,JEND= 4    0     1    32
   NPROC,MYID,JSTART,JEND= 4    3    97    128
 LOOP,ERR1= 2293    0.99947503029926 2768E-05
 X(M,J),J=1, 128 ,8
   .382E+00    .116E+00    .661E-01    .477E-01    .382E-01    .322E-01    .282E-01    .254E-01
   .232E-01     .215E-01    .202E-01    .190E-01    .181E-01    .173E-01    .167E-01    .161E-01
```

```
   MYID, ELAPSED TIME= 3    1.36376893613487482
   MYID, ELAPSED TIME= 1    1.36466074455529451
   MYID, ELAPSED TIME= 2    1.36448787245899439
   MYID, ELAPSED TIME= 0    1.36450297851115465
```

該平行版程式在 IBM SP2 SMP 八棵 CPU 上執行的結果如下。當 ITER 到達 946 次時其最大誤差值已經小於 EPSILON 而達到收斂的狀態，共耗時 0.67 秒。比原來四棵 CPU 的 2293 次又少了一半以上，收斂得更快。平行績效(speed up) =2.33/0.67=3.48 倍。

```
ATTENTION: 0031-408    8 tasks allocated by LoadLeveler, continuing...
   NPROC,MYID,JSTART,JEND= 8    0    1    16
   NPROC,MYID,JSTART,JEND= 8    1    17    32
   NPROC,MYID,JSTART,JEND= 8    6    97    112
   NPROC,MYID,JSTART,JEND= 8    7    113    128
   NPROC,MYID,JSTART,JEND= 8    2    33    48
   NPROC,MYID,JSTART,JEND= 8    4    65    80
   NPROC,MYID,JSTART,JEND= 8    3    49    64
   NPROC,MYID,JSTART,JEND= 8    5    81    96
LOOP, ERR1= 946      0.9993067578664693 86E-05
X(M,J),J=1, 128 ,8
   .382E+00    .116E+00    .661E-01    .478E-01    .381E-01    .322E-01    .282E-01    .254E-01
   .232E-01      .215E-01    .202E-01    .190E-01    .181E-01    .173E-01    .167E-01    .161E-01
   MYID, ELAPSED TIME= 1    0.667946914210915565
   MYID, ELAPSED TIME= 2    0.667559999972581863
   MYID, ELAPSED TIME= 3    0.667731808498501778
   MYID, ELAPSED TIME= 5    0.667331914417445660
   MYID, ELAPSED TIME= 4    0.667626276612281799
   MYID, ELAPSED TIME= 6    0.667829467914998531
   MYID, ELAPSED TIME= 7    0.667482553981244564
   MYID, ELAPSED TIME= 0    0.668705957010388374
```

8.3 斑馬線 SOR 解法

如果在第二維上做平行計算時，也可以採用斑馬線 SOR 解法來解決這一類的問題。請參閱圖 8.4，先計算所有白色元素之後，才計算黑色元素。不論計算白色元素時或計算黑色元素時都必須順著第一維的順序從 1 到 M 一路計算下去。

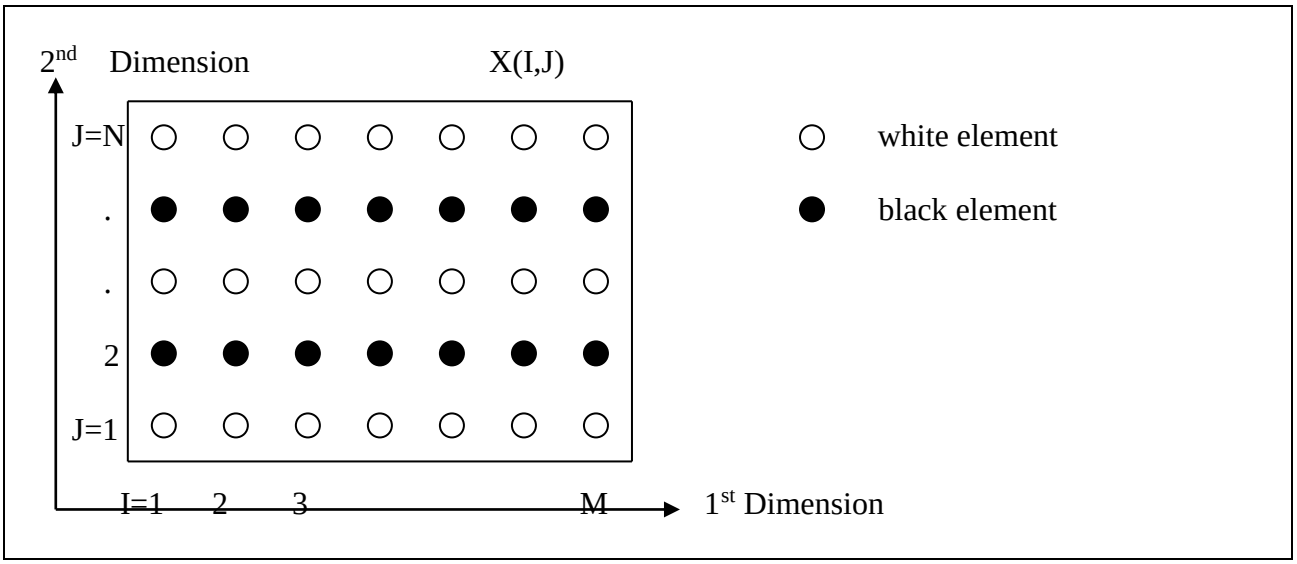


圖 8.4 斑馬線 SOR 解法示意圖

斑馬線 SOR 解法的循序程式如下

```
      PROGRAM SOR_ZEBRA
C
C     Sequential version of zebra SOR (Successive Over-Relaxation Method)
C
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (M=128, N=128)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,EPSILON,ABST,CLOCK
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE (IC)/DBLE (IR)
      OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
```

```
      READ(7)    X
      EPSILON=1.0D-5
      OMEGA=0.5
      DO 998 ITER=1,6000
        ERR1=0.0
        DO J=1,N,2
        DO I=1,M
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        DO J=2,N,2
        DO I=1,M
          TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        IF(ERR1.LE.EPSILON) GOTO 999
  998   CONTINUE
  999   CONTINUE
      WRITE(*,*) 'ITER, ERR1=', ITER, ERR1
      WRITE(*,*) 'X(M,J),J=1,', N, ',8'
      WRITE(*,101) (X(M,J),J=1,N,8)
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK=DBLE (IC)/DBLE (IR) - CLOCK
      PRINT *,' ELAPSED TIME=', CLOCK
  101 FORMAT(8E10.3)
      STOP
      END
```

循序程式 SOR_ZEBRA 在 IBM SP2 SMP 上測試的結果如下，當 ITER 到達 4899 次時誤差已經小於 EPSILON，耗時 6.02 秒。

```
ITER, ERR1= 4899    0.999871763466067542E-05
  X(M,J),J=1, 128 ,8
    .382E+00    .116E+00    .661E-01    .477E-01    .382E-01    .324E-01    .284E-01    .256E-01
    .234E-01    .217E-01    .204E-01    .192E-01    .183E-01    .175E-01    .167E-01    .161E-01
ELAPSED TIME= 6.02000000000407454
```

在第二維上做平行計算時，如圖 8.5，每一個 CPU 分配到的格點數必須為偶數，只有最後一個 CPU 可以是例外。所以各個 CPU 起迄 index 改用下面的方式來計算。

```
N2=(N+1)/2
CALL STARTEND(MYID,NPROC,1,N2,JSTART,JEND)
JSTART=JSTART*2-1
JEND=MIN(N, JEND*2)
```

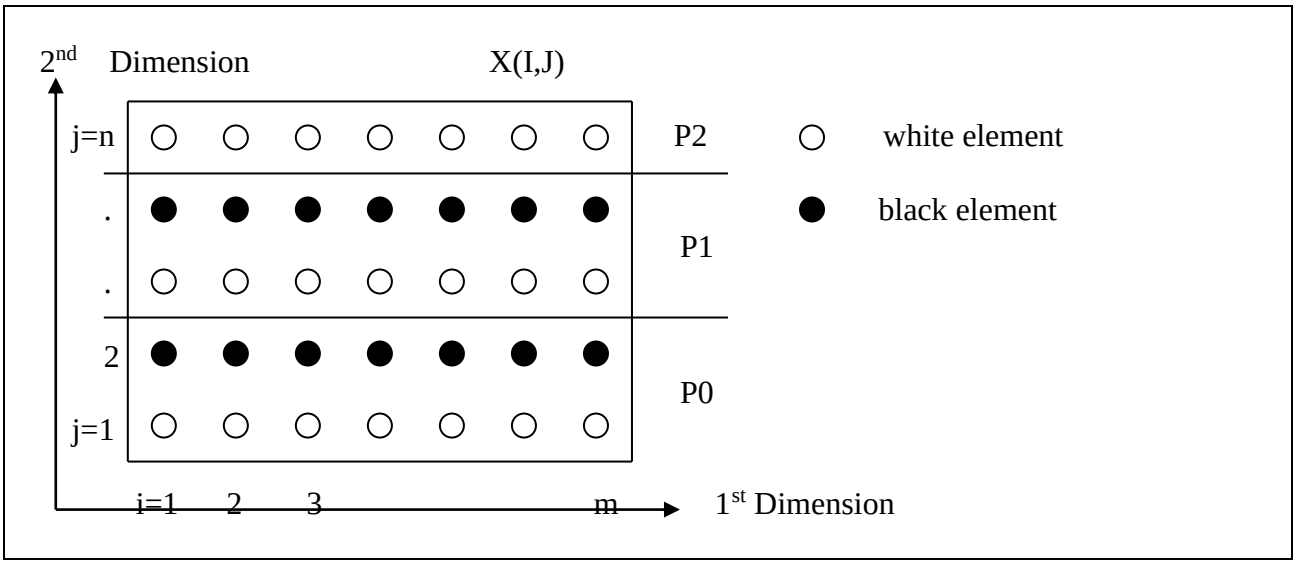


圖 8.5 斑馬線 SOR 解法平行計算示意圖

每個 CPU 都是先計算白色格點，也就是 index J 從 JSTART 開始，此時 JSTART-1 的資料在前一個 CPU 上，所以要在計算之前使用 MPI_SENDRECV 取得 JSTART-1 的一列資料。在計算白色格點時 J+1 的資料在各該 CPU 的範圍之內，不必外求。在計算黑色格點時要用到

JEND+1 的資料，所以在計算黑色格點之前使用 MPI_SENDRECV 取得 JEND+1 的一列資料。

在計算黑色格點時 J-1 的資料在各該 CPU 的範圍之內，不必外求。斑馬線 SOR_ZEBRAP 平行計算程式如下。

```
      PROGRAM SOR_ZEBRAP
C
C       Parallel version of zebra SOR (Successive Over-Relaxation Method)
C
      IMPLICIT REAL*8 (A-H,O-Z)
      INCLUDE     'mpif.h'
      PARAMETER (MM=129,NN=129,M=MM-1,N=NN-1)
      REAL*8    X(0:MM,0:NN),TEMP,OMEGA,ERR1,GERR1,EPS,ABST,CLOCK
C---------------------------------------------------------------
      INTEGER    NPROC, MYID, ISTATUS(MPI_STATUS_SIZE),J START, JEND,
     1                JSTARTM1, JENDP1, L_NBR, R_NBR
      COMMON/COMM/ NPROC, MYID, ISTATUS,J START, JEND, L_NBR, R_NBR,
     1                        LASTP, JSTARTG(0:31), JENDG(0:31)
C---------------------------------------------------------------
      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE    (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
      CALL MPI_BARRIER         (MPI_COMM_WORLD, IERR)
      CLOCK=MPI_WTIME()
      LASTP=NPROC-1
      N2=(N+1)/2
      DO I=0,LASTP
        CALL STARTEND (I,NPROC,1,N2,JSTART,JEND)
        JSTARTG(I)=JSTART*2-1
        JENDG(I)=MIN(N, JEND*2)
      ENDDO
      JSTART=JSTARTG(MYID)
      JEND=JENDG(MYID)
      PRINT *,' NPROC,MYID,JSTART,JEND=', NPROC,MYID,JSTART,JEND
      JSTARTM1=JSTART-1
      JENDP1=JEND+1
```

```
 L_NBR=MYID-1
 R_NBR=MYID+1
 IF(MYID.EQ.0) L_NBR=MPI_PROC_NULL
 IF(MYID.EQ.LASTP) R_NBR=MPI_PROC_NULL

 IF(MYID.EQ.0) THEN
    OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
    READ(7)    X
 ENDIF
 KOUNT=(2+M)*(2+N)
 CALL MPI_BCAST( X, KOUNT, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
 EPS=1.0D-5
 OMEGA=0.5
 DO 900 ITER=1,6000
    ERR1=0.0
    ITAG=10
    CALL MPI_SENDRECV (X(1,JEND),         M, MPI_REAL8, R_NBR, ITAG,
1                      X(1,JSTARTM1), M, MPI_REAL8, L_NBR, ITAG,
2                      MPI_COMM_WORLD, ISTATUS, IERR)
    DO J=JSTART,JEND,2
    DO I=1,M
       TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
       X(I,J)=X(I,J)+OMEGA*TEMP
       ABST=DABS(TEMP)
       IF(ABST.GT.ERR1) ERR1=ABST
    ENDDO
    ENDDO
    ITAG=20
    CALL MPI_SENDRECV (X(1,JSTART), M, MPI_REAL8, L_NBR, ITAG,
1                      X(1,JENDP1), M, MPI_REAL8, R_NBR, ITAG,
2                      MPI_COMM_WORLD, ISTATUS, IERR)
    DO J=JSTART+1, JEND, 2
    DO I=1,M
       TEMP=0.25*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )-X(I,J)
       X(I,J)=X(I,J)+OMEGA*TEMP
       ABST=DABS(TEMP)
       IF(ABST.GT.ERR1) ERR1=ABST
```

```
      ENDDO
      ENDDO
      CALL MPI_ALLREDUCE ( ERR1, GERR1, 1, MPI_REAL8, MPI_MAX,
     1                                    MPI_COMM_WORLD, IERR)
      ERR1=GERR1
      IF(ERR1 .LE. EPS) GOTO 910
  900 CONTINUE
  910 CONTINUE
      IF( MYID.EQ.0) THEN
        DO I=1,LASTP
          JSTART1=JSTARTG(I)
          KOUNT=(JENDG(I)-JSTART1+1)*M
          CALL MPI_RECV (X(1,JSTART1), KOUNT, MPI_REAL8, I, ITAG,
     1                                MPI_COMM_WORLD, ISTATUS, IERR)
        ENDDO
        WRITE(*,*) 'ITER, ERR1=', ITER, ERR1
        WRITE(*,*) 'X(M,J),J=1,', N, ',8'
        WRITE(*,101) ( X(M,J),J=1,N,8)
      ELSE
        KOUNT=(JEND-JSTART+1)*M
        CALL MPI_SEND (X(1,JSTART), KOUNT, MPI_REAL8, 0, ITAG,
     1                             MPI_COMM_WORLD, IERR)
      ENDIF
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,' MYID, ELAPSED TIME=', MYID, CLOCK
  101 FORMAT(8E10.3)
      CALL MPI_FINALIZE( MPI_ERR)
      STOP
      END
      SUBROUTINE STARTEND(MYID,NPROC,IS1,IS2,ISTART,IEND)
      INTEGER MYID,NPROC,IS1,IS2,ISTART,IEND
      ILENGTH=IS2-IS1+1
      IBLOCK=ILENGTH/NPROC
      IR=ILENGTH-IBLOCK*NPROC
      IF(MYID.LT.IR) THEN
        ISTART=IS1+MYID*(IBLOCK+1)
        IEND=ISTART+IBLOCK
      ELSE
```

```
      ISTART=IS1+MYID*IBLOCK+IR
      IEND=ISTART+IBLOCK-1
    ENDIF
    IF(ILENGTH.LT.1) THEN
      ISTART=1
      IEND=0
    ENDIF
    RETURN
    END
```

平行程式 SOR_ZEBRAP 在 IBM SP2 SMP 上四個 CPU 平行計算的結果如下。當 ITER 到達 4899 次時誤差已經小於 EPSILON，耗時 2.54 秒，平行績效(speed up) =6.02/2.54=2.37 倍。

、，。

```
ATTENTION: 0031-408    4 nodes allocated by LoadLeveler, continuing...
   NPROC,MYID,JSTART,JEND= 4    0     1     32
   NPROC,MYID,JSTART,JEND= 4    3    97    128
   NPROC,MYID,JSTART,JEND= 4    1    33    64
   NPROC,MYID,JSTART,JEND= 4    2    65    96
   MYID, ELAPSED TIME= 1    2.54448266047984362
   MYID, ELAPSED TIME= 2    2.54423265997320414
   MYID, ELAPSED TIME= 3    2.54461585078388453
  ITER, ERR1= 4899    0.999871763466067542E-05
  X(M,J),J=1, 128 ,8
   .382E+00    .116E+00    .661E-01    .477E-01    .382E-01    .324E-01    .284E-01    .256E-01
   .234E-01    .217E-01    .204E-01    .192E-01    .183E-01    .175E-01    .167E-01    .161E-01
   MYID, ELAPSED TIME= 0    2.54533478617668152
```

8.4 八方依賴與四色點間隔 SOR 解法

另外一種情況是陣列元素 x(i,j) 的值係由其左、右、上、下、左上、左下、右上、右下八個元素的值計算得來的，如圖 8.6 所示。其核心 DO loop 如下：

```
      ERR1=0.0
      DO J=1,N
        DO I=1,M
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) +X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
      ENDDO
```

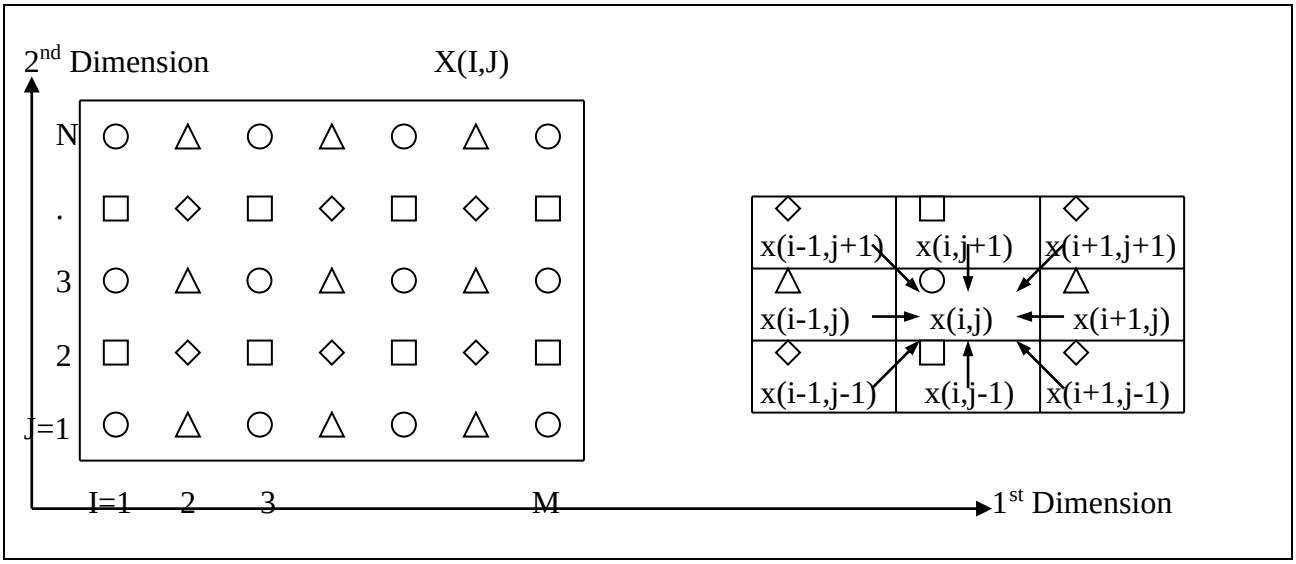


圖 8.6 八方依賴 SOR 解法示意圖

這一種程式不能夠採用黑白點間隔法來解，倒可以採用斑馬線法來解，此處介紹四色點解法。也就是利用四種顏色(此處用四種圖形)來代表相鄰的四個格點，如圖 8.6 所示。該圖中任何一

個格點的周圍八個格點都跟該格點相異，例如圓圈的周圍八個格點都不是圓圈，所以就可以先計算所有的圓圈，再計算所有的三角形，然後計算所有的方塊，最後計算所有的菱形。其循序程式如下。

```
      PROGRAM    COLOR_SOR
C
C       Sequential version of 4 color Successive Over-Relaxation Method
C
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (M=128,N=128)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,EPSILON,ABST
C
      CALL SYSTEM_CLOCK(IC,IR,IM)
      CLOCK= DBLE (IC)/ DBLE (IR)
      OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
      READ(7)    X
      EPSILON=1D-5
      OMEGA=0.5
      DO 900 ITER=1,6000
        ERR1=0.0
        DO J=1,N,2                    ! update circle grid
        DO I=1,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        DO J=1,N,2                    ! update triangle grid
        DO I=2,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
```

```
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        DO J=2,N,2                    ! update square grid
        DO I=1,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        DO J=2,N,2                    ! update <> grid
        DO I=2,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        IF(ERR1.LE.EPSILON) GOTO 901
  900   CONTINUE
  901   CONTINUE
        WRITE(*,*) 'ITER,ERR1=', ITER,ERR1
        WRITE(*,*) 'X(M,J),J=1,', N, ',8'
        WRITE(*,101) (X(M,J),J=1,N,8)
        CALL SYSTEM_CLOCK(IC,IR,IM)
        CLOCK=DBLE(IC)/ DBLE (IR) - CLOCK
        PRINT *,' ELAPSED TIME=', CLOCK
  101   FORMAT(8E10.3)
        STOP
        END
```

循序程式 COLOR_SOR 在 IBM SP2 SMP 執行的結果如下。當 ITER 到達 4171 次時其最大誤

差值已經小於 EPSILON 而達到收斂的狀態，共耗時 3.33 秒。

```
 ITER,ERR1= 4171    0.9996991088084827 84E-05
  X(M,J),J=1, 128 ,8
    .278E+00   .116E+00   .661E-01   .477E-01   .381E-01   .323E-01   .284E-01   .255E-01
    .234E-01   .217E-01   .203E-01   .192E-01   .183E-01   .175E-01   .168E-01   .161E-01
ELAPSED TIME= 3.32812500000000000
```

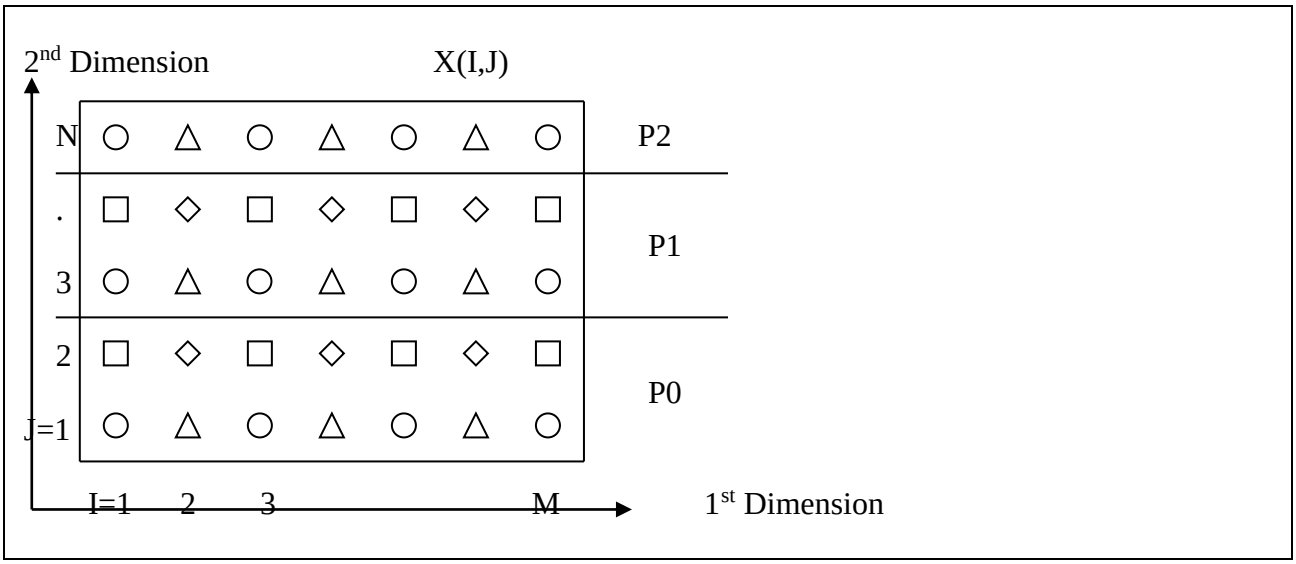


圖 8.6 八方依賴 SOR 解法平行計算示意圖

要將此程式平行化可在第二維上做計算切割，惟一的條件是每一個 CPU 分配到的格點數必須是偶數，只有最後一個 CPU 可以是例外。跟斑馬線法一樣各個 CPU 起迄 index 改用下面的方式來計算。

```
N2=(N+1)/2
CALL STARTEND(MYID,NPROC,1,N2,JSTART,JEND)
JSTART=JSTART*2-1
JEND=MIN(N, JEND*2)
```

在計算圓圈與三角形格點時要用到 JSTART-1 的資料，所以在這之前要先取得 JSTART-1 的一列資料。同理，計算方塊與菱形格點時要用到 JEND+1 的資料，所以在這之前要先取得 JEND+1 的一列資料。其他方向的資料都在各該 CPU 的範圍之內。

```
      PROGRAM COLOR_SORP
C
C       Parallel on 2nd dimension of 4 colour Successive Over-Relaxation Method
C
      IMPLICIT REAL*8 (A-H,O-Z)
      INCLUDE    'mpif.h'
      PARAMETER (M=128,N=128)
      REAL*8    X(0:M+1,0:N+1),TEMP,OMEGA,ERR1,EPSILON,ABST
C
      INTEGER    NPROC,MYID,ISTATUS(MPI_STATUS_SIZE),JSTART,JEND,
     1           JSTARTM1,JENDP1,L_NBR,R_NBR
      COMMON/NPROC, MYID, ISTATUS, JSTART, JEND, L_NBR, R_NBR, LASTP,
     1           JSTARTG(0:31), JENDG(0:31)
C-----------------------------------------------------------------

      CALL MPI_INIT (IERR)
      CALL MPI_COMM_SIZE    (MPI_COMM_WORLD, NPROC, IERR)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD,    MYID, IERR)
      CALL MPI_BARRIER         (MPI_COMM_WORLD, IERR)
      CLOCK=MPI_WTIME()
      LASTP=NPROC-1
      N2=(N+1)/2
      CALL STARTEND (NPROC,1,N2,JSTARTG,JENDG)
      JSTART=JSTARTG(MYID)
      JEND=JENDG(MYID)
      PRINT *,' NPROC,MYID,JSTART,JEND=', NPROC,MYID,JSTART,JEND
      JSTARTM1=JSTART-1
      JENDP1=JEND+1

      L_NBR=MYID-1
      R_NBR=MYID+1
```

```
      IF(MYID.EQ.0)          L_NBR=MPI_PROC_NULL
      IF(MYID.EQ.LASTP) R_NBR=MPI_PROC_NULL

      IF(MYID.EQ.0) THEN
        OPEN (7,FILE='input.dat',FORM='UNFORMATTED')
        READ(7)    X
      ENDIF
      KOUNT=(2+M)*(2+N)
      CALL MPI_BCAST( X, KOUNT, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
      EPSILON=1.0D-5
      OMEGA=0.5
      DO 900 ITER=1,6000
        ERR1=0.0
        ITAG=10
        CALL MPI_SENDRECV (X(1,JEND),         M, MPI_REAL8, R_NBR, ITAG,
     1                                  X(1,JSTARTM1), M, MPI_REAL8, L_NBR, ITAG,
     2                                  MPI_COMM_WORLD, ISTATUS, IERR)
CCC      DO J=1,N,2                      ! update circle
        DO J=JSTART,JEND,2
        DO I=1,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
CCC     DO J=1,N,2                      ! update triangle grid
        DO J=JSTART,JEND,2
        DO I=2,M,2
          TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1                +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
          X(I,J)=X(I,J)+OMEGA*TEMP
          ABST=ABS(TEMP)
          IF(ABST.GT.ERR1) ERR1=ABST
        ENDDO
        ENDDO
        ITAG=10
```

```
      CALL MPI_SENDRECV (X(1,JSTART), M, MPI_REAL8, L_NBR, ITAG,
     1                   X(1,JENDP1), M, MPI_REAL8, R_NBR, ITAG,
     2                   MPI_COMM_WORLD, ISTATUS, IERR)
CCC   DO J=2,N,2                ! update square grid
      DO J=JSTART+1,JEND,2
      DO I=1,M,2
        TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1             +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
        X(I,J)=X(I,J)+OMEGA*TEMP
        ABST=ABS(TEMP)
        IF(ABST.GT.ERR1) ERR1=ABST
      ENDDO
      ENDDO
CCC   DO J=2,N,2                ! update <> grid
      DO J=JSTART+1,JEND,2
      DO I=2,M,2
        TEMP=0.125*( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+X(I-1,J-1)
     1             +X(I-1,J+1)+X(I+1,J-1)+X(I+1,J+1) ) -X(I,J)
        X(I,J)=X(I,J)+OMEGA*TEMP
        ABST=ABS(TEMP)
        IF(ABST.GT.ERR1) ERR1=ABST
      ENDDO
      ENDDO
      IF(ERR1.LE.EPSILON) GOTO 910
  900 CONTINUE
  910 CONTINUE
      IF(MYID.EQ.0) THEN
        DO ISRC=1,LASTP
          JSTART1=JSTARTG(ISRC)
          KOUNT=(JENDG(ISRC)-JSTART1+1)*M
          CALL MPI_RECV (X(1,JSTART1), KOUNT, MPI_REAL8, ISRC, ITAG,
     1                   MPI_COMM_WORLD, ISTATUS, IERR)
        ENDDO
        WRITE(*,*) 'ITER, ERR1=', ITER, ERR1
        WRITE(*,*) 'X(M,J),J=1,', N, ',8'
        WRITE(*,101) (X(M,J),J=1,N,8)
      ELSE
        KOUNT=(JEND-JSTART+1)*M
```

```
      CALL MPI_SEND (X(1,JSTART), KOUNT, MPI_REAL8, 0, ITAG,
     1                           MPI_COMM_WORLD, IERR)
      ENDIF
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,' ELAPSED TIME=', CLOCK
  101 FORMAT(8E10.3)
      CALL MPI_FINALIZE (MPI_ERR)
      STOP
      END
      SUBROUTINE STARTEND(NPROC,IS1,IS2,ISTART,IEND,ICOUNT)

      INTEGER   NPROC,IS1,IS2,ISTART(0:31),IEND(0:31), ICOUNT(0:31)

      ILENGTH=IS2-IS1+1

      IBLOCK=ILENGTH/NPROC

      IR=ILENGTH-IBLOCK*NPROC

      DO I=0,NPROC-1

        IF(I.LT.IR) THEN

          ISTART(I)=IS1+I*(IBLOCK+1)

          IEND(I)=ISTART(I)+IBLOCK

        ELSE

          ISTART(I)=IS1+I*IBLOCK+IR

          IEND(I)=ISTART(I)+IBLOCK-1

        ENDIF

        IF(ILENGTH.LT.1) THEN

          ISTART(I)=1
```

```
            IEND(I)=0
          ENDIF
          ICOUNT(I)=IEND(I)-ISTART(I)+1
        ENDDO
        RETURN
        END
```

平行程式 COLOR_SORP 在 IBM SP2 SMP 上四個 CPU 執行的結果如下。當 ITER 到達 4171 次時其最大誤差值已經小於 EPSILON 而達到收斂的狀態，共耗時 1.76 秒。平行績效(speed up) =3.33/1.76=1.89 倍。

```
 NPROC,MYID,JSTART,JEND= 4   0   1   32
 NPROC,MYID,JSTART,JEND= 4   2   65   96
 NPROC,MYID,JSTART,JEND= 4   1   33   64
 NPROC,MYID,JSTART,JEND= 4   3   97   128
 MYID, ELAPSED TIME= 1   1.75946882925927639
 MYID, ELAPSED TIME= 2   1.75943914894014597
 MYID, ELAPSED TIME= 3   1.75958851072937250
ITER,ERR1= 4171   0.9996991088084827 84E-05
X(M,J),J=1, 128 ,8
 .278E+00   .116E+00   .661E-01   .477E-01   .381E-01   .323E-01   .284E-01   .255E-01
 .234E-01   .217E-01   .203E-01   .192E-01   .183E-01   .175E-01   .168E-01   .161E-01
 MYID, ELAPSED TIME= 0   1.7604879783466458
```

第九章　有限元素法程式

本章將討論有限元素法 (finite element method - FEM) 程式的平行解法。有限元素法資料依賴關係比有限差異法 (finite difference method) 無規則性，一般說來也比較難以平行化。有限元素隱性解法(implicit method)的重點工作落在解稀疏對稱線性方程式。本章將討論有限元素法的顯性解法 (explicit method) 。

第一節 有限元素法的循序程式。

第二節 有限元素法的平行程式。

9.1 有限元素法的循序程式

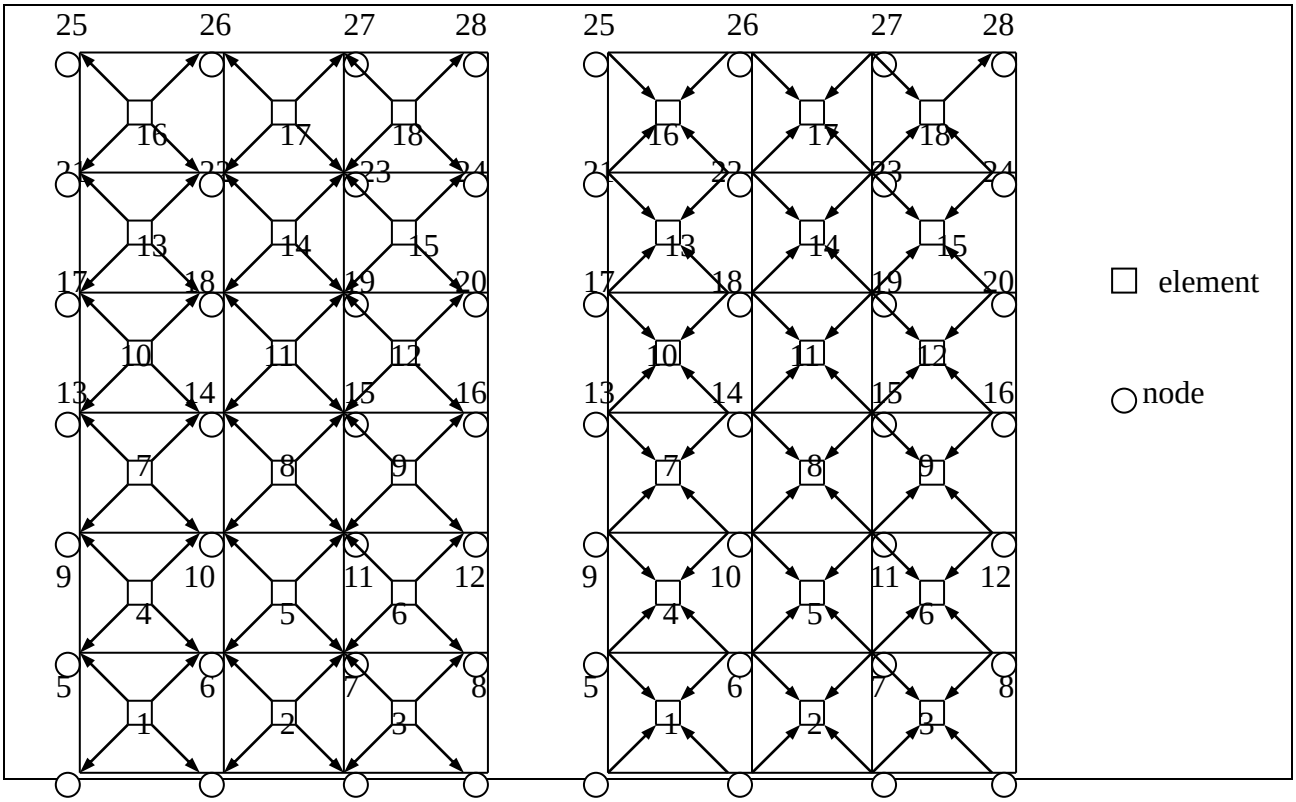
下面是一個簡化了的有限元素法循序程式，模擬有限元素顯性解法的主要資料依賴關係。

```
      PROGRAM    FEM_SEQ
      PARAMETER (NE=18, NN=28)
      REAL*8      VE(NE), VN(NN)
      INTEGER    INDEX(4,NE)

      OPEN (1, FILE='index.dat' )
      DO IE=1,NE
         READ (1,*) (INDEX(J,IE), J=1,4 )
      ENDDO
      DO IE=1,NE
        VE(IE)=10.0*IE
      ENDDO
      DO IN=1,NN
        VN(IN)=100.0*IN
      ENDDO
      DO 100 ITIME=1, 10
        DO 10 IE=1,NE
          DO J=1,4
            VN(INDEX(J,IE)) = VN(INDEX(J,IE)) + VE(IE)
          ENDDO
10      CONTINUE
        DO 20 IN=1, NN
          VN(IN) = VN(IN) * 0.25
20      CONTINUE
        DO 30 IE=1,NE
          DO J=1,4
            VE(IE) = VE(IE) + VN( INDEX(J,IE) )
          ENDDO
30      CONTINUE
        DO 40 IE=1, NE
          VE(IE) = VE(IE) *0.25
40      CONTINUE
```

```
100   CONTINUE
      PRINT *, 'Result of VN'
      PRINT 101, VN
      PRINT *, 'Result of VE'
      PRINT 101, VE
101   FORMAT(8F10.3)
      STOP
      END
```

Element 跟 node 的關聯如圖 9.1 所示。共有 18 個 element (方塊) ，每個 element 跟四個 node (圓圈) 相連接。這一種方法也適用於 'non-structured grid' (unstructured grid)。上面這一個程式裏的陣列 VE 和 VN 分別代表 element 和 node 上的某些數量。在 ITIME loop 裏共有四個 loop 分別計算 VE 和 VN 的值。在 Loop-10 是以 VE 的值加到跟它相關的四個 VN 裏，在 loop-20 是各個 VN 值各自更新，在 loop-30 是以更新過的 VN 值加到與其相關的 VE 上，在 loop-40 是各個 VE 值各自更新。



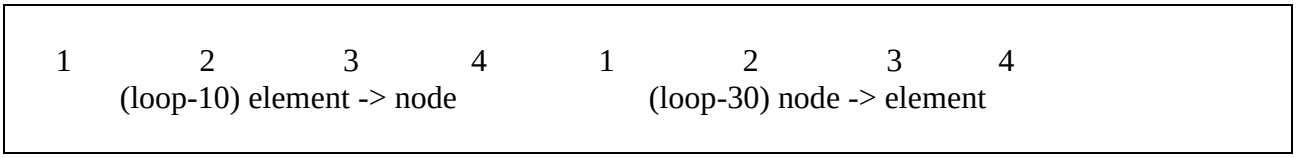


圖 9.1 有限元素法 element 跟 node 關聯圖

與每一個 element 相關聯的 node 編號則是存放在陣列 INDEX 裏。INDEX(I,J)的值代表第 J 個 element 相鄰的四個 node 的編號，I 值是從 1 到 4。

$$INDEX=\begin{bmatrix} 1 & 2 & 3 & 5 & 6 & 7 & 9 & 10 & 11 & 13 & 14 & 15 & 17 & 18 & 19 & 21 & 22 & 23 \\ 2 & 3 & 4 & 6 & 7 & 8 & 10 & 11 & 12 & 14 & 15 & 16 & 18 & 19 & 20 & 22 & 23 & 24 \\ 6 & 7 & 8 & 10 & 11 & 12 & 14 & 15 & 16 & 18 & 19 & 20 & 22 & 23 & 24 & 26 & 27 & 28 \\ 5 & 6 & 7 & 9 & 10 & 11 & 13 & 14 & 15 & 17 & 18 & 19 & 21 & 22 & 23 & 25 & 26 & 27 \end{bmatrix}$$

循序程式 FEM_SEQ 之執行結果如下

```
Result of VN
   303.506   737.138   743.620   309.989   905.479  2197.706  2214.970   922.743
  1476.091  3579.268  3602.639  1499.462  1927.588  4670.236  4695.415  1952.767
  2066.994  5005.284  5028.654  2090.365  1655.717  4008.155  4025.419  1672.981
   642.193  1554.410  1560.892   648.676
Result of VE
  1281.497  1823.797  1298.016  2526.136  3591.987  2554.243  3617.916  5139.575
  3651.343  4262.652  6051.210  4296.079  3991.965  5664.587  4020.072  2474.166
  3510.129  2490.686
```

9.2 有限元素法的平行程式

有限元素法的平行計算切割方式可就 element 依區塊切割法加以切割，如圖 9.2。18 個 element 在三棵 CPU 上平行時，每棵 CPU 負責處理 6 個 element 及其相鄰的 node。介於兩棵 CPU 邊界上的 node 則以 rank 較小的 CPU 為主要的處理器(primary processor)，以 rank 較大的 CPU 為次要的處理器(secondary processor)，用虛線圓圈來代表。

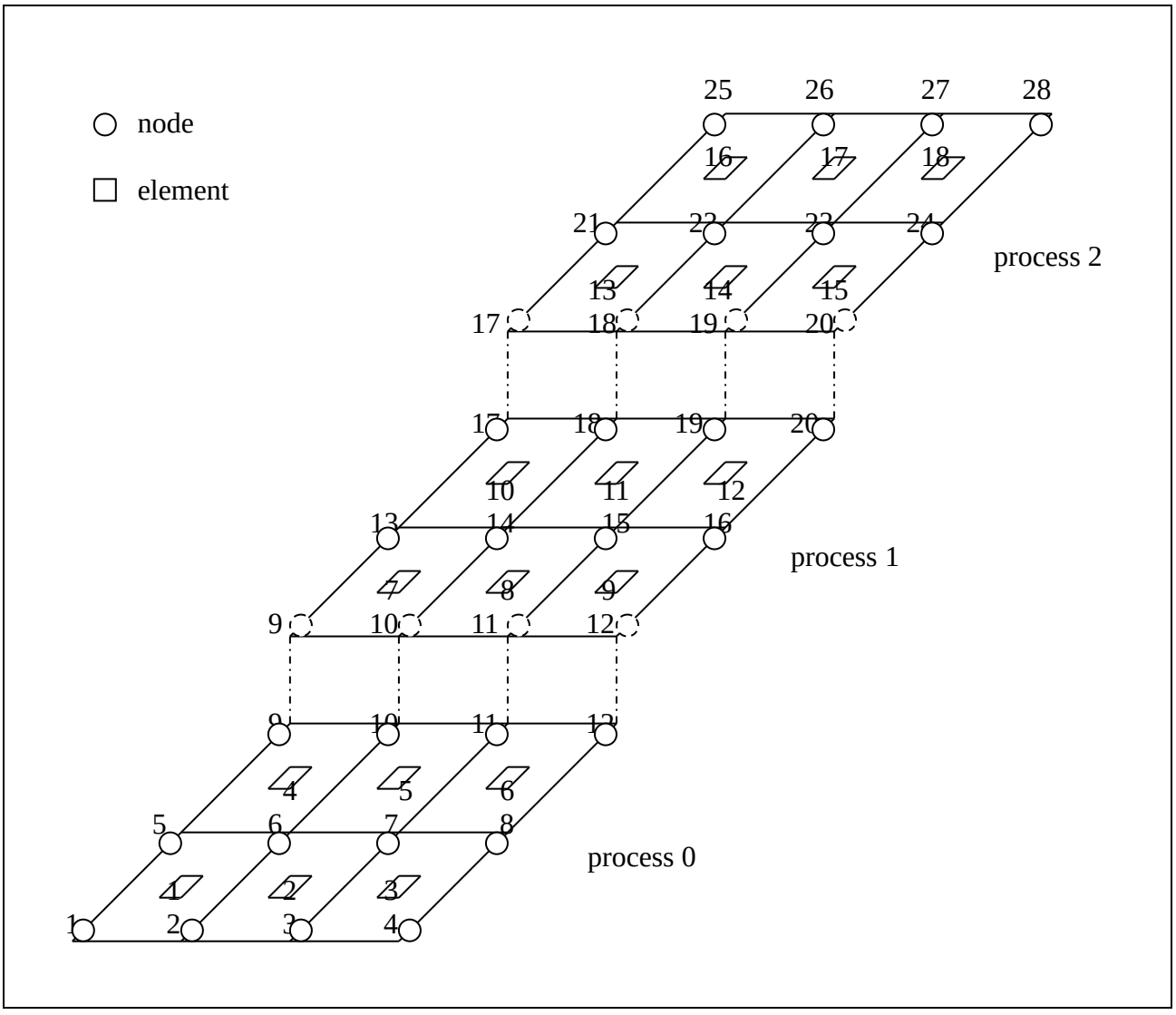


圖 9.2 有限元素法平行計算的資料切割圖

由於 element 是採用區段切割，需要兩個整數陣列來存放各個 CPU 的 element 數量和起點，IECNTG(I) 存放 CPU I 的 element 數量，IESTARTG(I) 存放 CPU I 的 element 起點。另外還需要兩個整數陣列來存放各個 CPU 的主要 node 數量及其編號，INCNTG(I) 存放 CPU I 的主要 node 數量，INODEG(J,I) 存放 CPU I 的主要 node 編號。如圖 9.3，字尾 G 代表 Global。

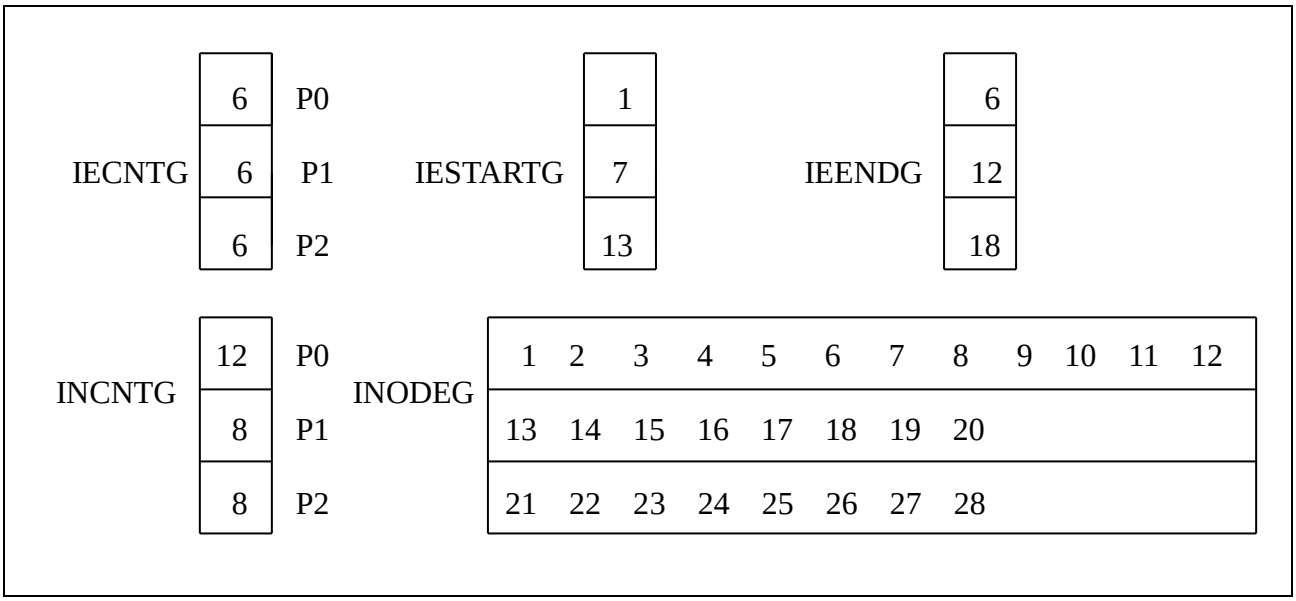


圖 9.3　element 及 node 分割資料結構圖

接下來要考慮與鄰居 CPU 共用一個 node 的問題。這一個特例一個 CPU 只要考慮其左右鄰居就已足夠，但若對於一般非結構化格點(或叫做不規則網格 irregular mesh)，一個 CPU 有可能與兩個以上的 CPU 相鄰，所以每一個 CPU 都要考慮與其他各個 CPU 之間的關係。此處需要兩組資料，一組用來存放該 CPU 與其他 CPU I 之間的次要 node 的數目 ISCNT (I)及其編號 ISNODE (J,I)，J=1,ISCNT (I)，另外一組用來存放該 CPU 與其他 CPU I 之間的主要 node 數目 IPCNT (I)及其編號 IPNODE (J,I)，J=1,IPCNT (I) 。這一個例子的上述資料如圖 9.4。

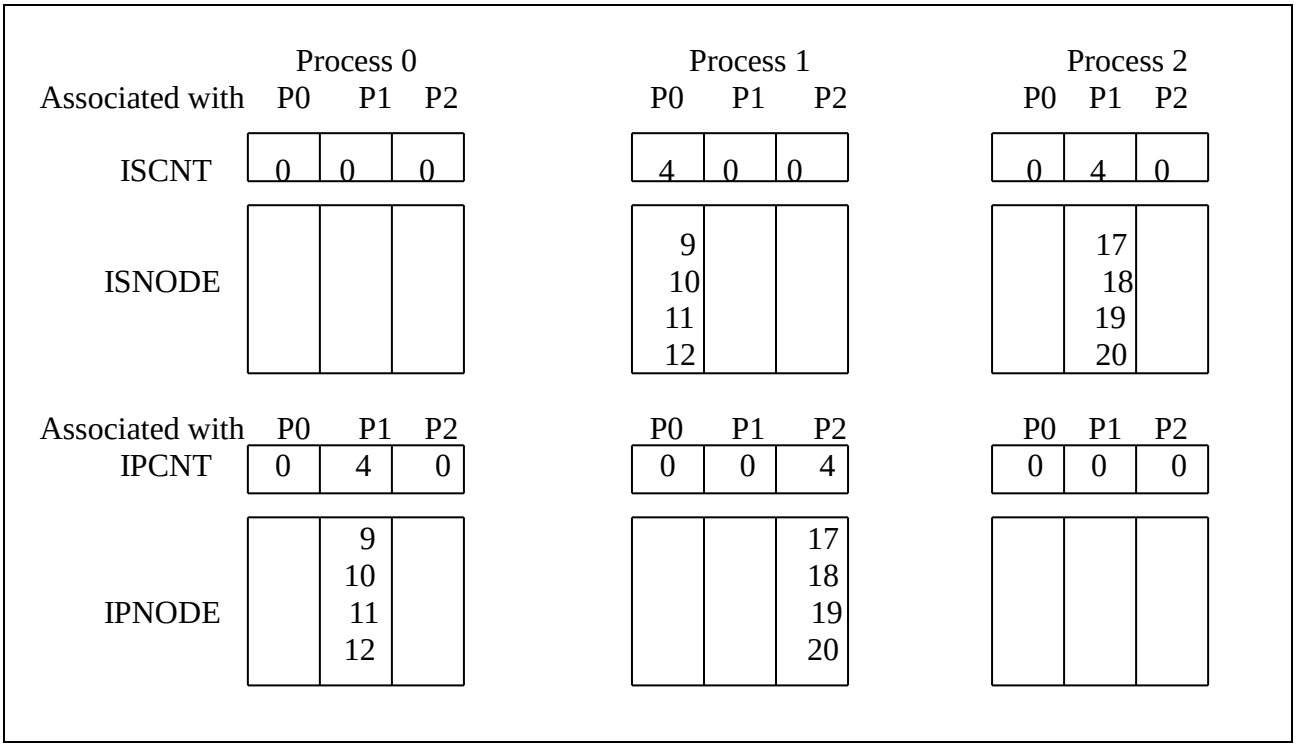


圖 9.4　CPU 間共用 node 之主副 CPU 關係圖

循序程式 FEM_SEQ 平行化之後如下，將在重要段落之前插入必要的說明。

```
      PROGRAM    FEM_PAR
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (NE=18, NN=28)
      DIMENSION    VE(NE), VN(NN)
      INTEGER       INDEX(4,NE)
      INCLUDE      'mpif.h'
      INTEGER       NPROC, MYID, ISTATUS(MPI_STATUS_SIZE)
      PARAMETER (NP=3)
C
C       global arrays
C
      COMMON/G/ IECNTG(0:NP-1), IESTARTG(0:NP-1), IEENDG(0:NP-1),
     1                  INCNTG(0:NP-1), INODEG(NN,0:NP-1), ITEMP(NN,NP-1)
C
C       local arrays
C
      COMMON/L/ISCNT(0:NP-1), ISNODE(NN,0:NP-1),
     1                  IPCNT(0:NP-1), IPNODE(NN,0:NP-1),
```

```
      2                BUFS(NN,0:NP-1), BUFR(NN,0:NP-1)

       CALL MPI_INIT (IERR)
       CALL MPI_COMM_SIZE    (MPI_COMM_WORLD, NPROC, IERR)
       CALL MPI_COMM_RANK (MPI_COMM_WORLD, MYID, IERR)
       CALL MPI_BARRIER          (MPI_COMM_WORLD, IERR)
       IF(NPROC.NE.NP) THEN
          IF(MYID.EQ.0) PRINT *,'NPROC .NE.',NP,'PROGRAM WILL STOP'
          CALL MPI_FINALIZE (MPI_ERR)
          STOP
       ENDIF
       CLOCK=MPI_WTIME()
       LASTP=NPROC-1
       IF (MYID.EQ.0) THEN
          OPEN (1, FILE='index.dat' )
          DO IE=1,NE
              READ (1,*) (INDEX(J,IE), J=1,4 )
          ENDDO
       ENDIF
       ICOMM=MPI_COMM_WORLD
       KOUNT = NE*4
       CALL MPI_BCAST (INDEX, KOUNT, MPI_INTEGER, 0, ICOMM, IERR)
C
C       clear count, CPU and node association indicator
C
       DO IRANK=0,LASTP
          INCNTG(IRANK)=0
          ISCNT(IRANK)=0
          IPCNT(IRANK)=0
          DO J=1,NN
             ITEMP(J,IRANK)=0
             ISNODE(J,IRANK)=0
             IPNODE(J,IRANK)=0
          ENDDO
       ENDDO
```

陣列 ITEMP 是用來記錄每一個 CPU 與那些 node 有關聯。上面這一個 DO loop 把每一個 CPU

所屬 NN 個 node 對應的位置清為零，在下面這一個 DO loop 把與各該 CPU 有關聯 node 的對應位置設為 1，參考圖 9.2 其結果如表 9.1 所示。

J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
P0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
P2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

表 9.1　ITEMP 設定後的內容

```
C
C       set node association indicator for associated nodes
C
        DO IRANK=0,LASTP
           CALL STARTEND (IRANK,NPROC,1,NE,IESTART,IEEND)
           IESTARTG(IRANK)=IESTART
           IEENDG(IRANK)=IEEND
           IECNTG(IRANK)=IEEND-IESTART+1
           DO IE=IESTART,IEEND
              DO J=1,4
                 ITEMP( INDEX(J,IE),IRANK) = 1
              ENDDO
           ENDDO
        ENDDO
        IESTART=IESTARTG(MYID)
        IEEND=IEENDG(MYID)
        PRINT *,' NPROC,MYID,IESTART,IEEND=', NPROC,MYID,IESTART,IEEND
```

由表 9.1 即可看出，當同一個 node 位置出現兩個(或兩個以上) 的 1 時，表示該兩個 CPU 共用該 node，例如 P0 和 P1 共用 node 9、10、11、12，這幾個 node 是 P0 的 primary node，也是 P1 的 secondary node。P1 和 P2 共用 node 17、18、19、20，這幾個 node 是 P1 的 primary node，也是 P2 的 secondary node。下面這一個 DO loop 就是做這種查核，並計數共用 node 的個數

ISCNT、IPCNT 及儲存共用 node 的代號於 ISNODE、IPNODE。這幾個變數名稱的第二個字母 S 代表 secondary，P 代表 primary。

```
C
C       count and store boundary node code
C
        DO IN=1,NN
          IFIRST=1
          DO IRANK=0,LASTP
            IF( ITEMP(IN,IRANK).EQ.1) THEN      ! node(in) belong to irank
              IF (IFIRST.EQ.1) THEN             ! 1st time itemp(in,irank)==1
                IFIRST=2
                II=IRANK
              ELSE                              ! 2nd time itemp(in,irank)==1
                ITEMP(IN,IRANK)=0               ! not irank's primary node, clear it
                IF (IRANK.EQ.MYID) THEN
                  ISCNT(II)=ISCNT(II)+1         ! secondary node count
                  ISNODE (ISCNT(II), II) = IN   ! secondary node code
                ELSEIF (II.EQ.MYID) THEN
                  IPCNT(IRANK)=IPCNT(IRANK)+1       ! primary node count
                  IPNODE (IPCNT(IRANK),IRANK)=IN    ! primary node code
                ENDIF
              ENDIF
            ENDIF
          ENDDO
        ENDDO
C
C       count and store all primary node code which belongs to each CPU
C
        DO IRANK=0, LASTP
          DO IN=1, NN
            IF (ITEMP(IN,IRANK) .EQ. 1) THEN
              INCNTG (IRANK) = INCNTG(IRANK) + 1
              INODEG (INCNTG(IRANK), IRANK) = IN
            ENDIF
          ENDDO
```

```
      ENDDO
C                                set initial values
      DO IE=1,NE
        VE(IE)=10.0*IE
      ENDDO
      DO IN=1,NN
        VN(IN)=100.0*IN
      ENDDO
```

下面開始 ITIME loop。在開始計算 VN 之前必須把次要 node 之 VN 值清為零，使得這些 node 算出來的 VN 值僅只代表次要 node 的貢獻值，然後把這些數值傳送給主要 node，與主要 node 部分相加。

```
      DO 900 ITIME=1, 10
        DO IRANK=0, LASTP
          DO IS=1,ISCNT(IRANK)
            VN(ISNODE(IS,IRANK)) = 0.0
          ENDDO
        ENDDO
CCC     DO 10 IE=1,NE
        DO 10 IE=IESTART,IEEND
          DO J=1,4
            VN(INDEX(J,IE))=VN(INDEX(J,IE)) + VE(IE)
          ENDDO
   10   CONTINUE
        DO IRANK=0, LASTP
          DO IS=1,ISCNT(IRANK)
            BUFS(IS,IRANK) = VN( ISNODE(IS,IRANK) )
          ENDDO
        ENDDO
        ITAG=10
        DO IRANK=0, LASTP
          IF (ISCNT(IRANK).GT.0)
     1      CALL MPI_SEND (BUFS(1,IRANK), ISCNT(IRANK), MPI_REAL8,
     2                     IRANK, ITAG, ICOMM, IERR)
```

```
      IF (IPCNT(IRANK).GT.0)
     1   CALL MPI_RECV (BUFR(1,IRANK), IPCNT(IRANK), MPI_REAL8,
     2                  IRANK, ITAG, ICOMM, ISTATUS, IERR)
      ENDDO
      DO IRANK=0, LASTP
        DO IP=1,IPCNT(IRANK)
          VN(IPNODE(IP,IRANK))=VN(IPNODE(IP,IRANK))+BUFR(IP,IRANK)
        ENDDO
      ENDDO
```

下面這一段 VN 值只在主要 CPU 裏更新，得到的新數值再傳送給次要 CPU。各個 CPU 再使用 VN 值來計算 VE 值(loop 30) ，最後各個 VE 值再各自更新(loop 40) 。

```
CCC   DO 20 IN=1, NN
      DO 20 II=1, INCNTG(MYID)
        IN = INODEG(II,MYID)
        VN(IN) = VN(IN) * 0.25
   20 CONTINUE
      DO IRANK=0, LASTP
        DO I=1,IPCNT(IRANK)
          BUFS(I,IRANK) = VN(IPNODE(I,IRANK))
        ENDDO
      ENDDO
      ITAG=20
      DO IRANK=0, LASTP
        IF (IPCNT(IRANK).GT.0)
     1   CALL MPI_SEND (BUFS(1,IRANK), IPCNT(IRANK), MPI_REAL8,
     2                  IRANK, ITAG, ICOMM, IERR)
        IF (ISCNT(IRANK).GT.0)
     1   CALL MPI_RECV (BUFR(1,IRANK), ISCNT(IRANK), MPI_REAL8,
     2                  IRANK, ITAG, ICOMM, ISTATUS, IERR)
      ENDDO
      DO IRANK=0, LASTP
        DO I=1,ISCNT(IRANK)
          VN( ISNODE(I,IRANK) ) = BUFR(I,IRANK)
        ENDDO
```

```
          ENDDO
CCC       DO 30 IE=1,NE
          DO 30 IE=IESTART,IEEND
            DO J=1,4
              VE(IE) = VE(IE) + VN(INDEX(J,IE))
            ENDDO
   30     CONTINUE
CCC       DO 40 IE=1, NE
          DO 40 IE=IESTART,IEEND
            VE(IE) = VE(IE) *0.25
   40     CONTINUE
  900 CONTINUE
```

在 loop 900 之後，把各個 CPU 計算出來的片段結果集中到 CPU 0，然後由 CPU 0 把整個陣列列印出來。

```
      DO I=1, INCNTG(MYID)
        BUFS(I,MYID) = VN(INODEG(I,MYID))
      ENDDO
      ITAG=30
      IF (MYID.EQ.0) THEN
        DO IRANK=1, LASTP
          CALL MPI_RECV (BUFR(1,IRANK), INCNTG(IRANK), MPI_REAL8,
     1                   IRANK, ITAG, ICOMM, ISTATUS, IERR)
        ENDDO
      ELSE
        CALL MPI_SEND (BUFS(1,MYID), INCNTG(MYID), MPI_REAL8,
     1                 0, ITAG, ICOMM, IERR)
      ENDIF
      IF (MYID.EQ.0) THEN
        DO IRANK=1, LASTP
          DO I=1, INCNTG(IRANK)
            VN (INODEG(I,IRANK)) = BUFR(I,IRANK)
          ENDDO
        ENDDO
      ENDIF
```

```
      ITAG=40
      IF (MYID.EQ.0) THEN
        DO IRANK=1, LASTP
          CALL MPI_RECV (VE(IESTARTG(IRANK)),IECNTG(IRANK),MPI_REAL8,
     1                   IRANK, ITAG, ICOMM, ISTATUS, IERR)
        ENDDO
      ELSE
        CALL MPI_SEND (VE(IESTART), IECNT, MPI_REAL8,
     1                 0, ITAG, ICOMM, MPI_ERR)
      ENDIF
      IF (MYID.EQ.0) THEN
        PRINT *, 'Result of VN'
        PRINT 101, VN
        PRINT *, 'Result of VE'
        PRINT 101, VE
      ENDIF
  101 FORMAT(8F10.3)
      CLOCK=MPI_WTIME() - CLOCK
      PRINT *,'MYID,CLOCK=', MYID,CLOCK
      CALL MPI_FINALIZE (MPI_ERR)
      STOP
      END
      SUBROUTINE STARTEND(ITASK,NUMTASK,IS1,IS2,ISTART,IEND)
      INTEGER ITASK,NUMTASK,IS1,IS2,ISTART,IEND
      ILENGTH=IS2-IS1+1
      IBLOCK=ILENGTH/NUMTASK
      IR=ILENGTH-IBLOCK*NUMTASK
      IF (ITASK .LT. IR) THEN
        ISTART=IS1+ITASK*(IBLOCK+1)
        IEND=ISTART+IBLOCK
      ELSE
        ISTART=IS1+ITASK*IBLOCK+IR
        IEND=ISTART+IBLOCK-1
      ENDIF
      IF( ILENGTH .LT. 1) THEN
        ISTART=1
        IEND=0
      ENDIF
```

```
      RETURN
      END
```

平行程度 FEM_PAR 在 IBM SP2 SMP 的三顆 CPU 上執行的結果如下，與循序程式 FEM_SEQ 計算出來的結果完全相同。

```
ATTENTION: 0031-408   3 tasks allocated by LoadLeveler, continuing...
   NPROC,MYID,IESTART,IEEND= 3   2   13   18
   NPROC,MYID,IESTART,IEEND= 3   1   7   12
   NPROC,MYID,IESTART,IEEND= 3   0   1   6
 MYID,CLOCK= 1   0.3008531965E-01
 MYID,CLOCK= 2   0.3012776561E-01
Result of VN
    303.506    737.138    743.620    309.989    905.479   2197.706   2214.970    922.743
   1476.091   3579.268   3602.639   1499.462   1927.588   4670.236   4695.415   1952.767
   2066.994   5005.284   5028.654   2090.365   1655.717   4008.156   4025.419   1672.981
    642.193   1554.410   1560.892    648.676
Result of VE
   1281.497   1823.797   1298.016   2526.136   3591.987   2554.243   3617.916   5139.575
   3651.343   4262.652   6051.210   4296.079   3991.965   5664.587   4020.072   2474.166
   3510.129   2490.686
 MYID,CLOCK= 0   0.3047159500E-01
```

附錄一 撰寫 C 語言的 MPI 程式

撰寫 C 語言的 MPI 程式要注意下列事項。

1. include file 是 mpi.h

2. MPI 用字的大小寫攸關重要(case sensitive) ，所有 function name 都是 MPI_Function 的樣子，其中的 MPI 和 function name 的第一個字母是大寫，其餘的都是小寫。例如 MPI_Init、MPI_Comm_size、MPI_Comm_rank、MPI_Finalize、MPI_Send、MPI_Recv 等等。而在 mpi.h 裏已經設定的 MPI 常數則全部都是大寫，例如 MPI_FLOAT、MPI_INT、MPI_DOUBLE、MPI_COMM_WORLD、MPI_SUM、MPI_MAX 等等。

3. MPI function 的引數(argument) 若為資料位址(address)時必須是 pointer。

4. 叫用 function 的 return code 是個整數，通常略而不用。

5. C 語言常用的 MPI 基本資料類別如下表所示

MPI data type	C data type	description
MPI_CHAR	signed char	1-byte character
MPI_SHORT	signed short iny	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		

6. MPI 縮減運作及相關的 C 語言 MPI 資料類別

MPI data type	C data type
MPI_SUM MPI_PROD MPI_MAX MPI_MIN	MPI_INT, MPI_SHORT, MPI_LONG, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE
MPI_MAXLOC MPI_MINLOC	MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT
MPI_LAND MPI_LOR MPI_LXOR	MPI_SHORT, MPI_LONG, MPI_INT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
MPI_BAND MPI_BOR MPI_BXOR	MPI_SHORT, MPI_LONG, MPI_INT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG

其中 MPI_MAXLOC 和 MPI_MINLOC 所用到的資料類別是 C 語言的 structure 如下表

Data type	Description (C structure)
MPI_FLOAT_INT	{MPI_FLOAT, MPI_INT}
MPI_DOUBLE_INT	{MPI_DOUBLE, MPI_INT}
MPI_LONG_INT	{MPI_LONG, MPI_INT}
MPI_2INT	{MPI_INT, MPI_INT}

7. 至於 STATUS、REQUEST 等在 Fortran 程式裏只要設定為整數或整數陣列即可，在 C 程式裏則設定如下

MPI_Status	status
MPI_Request	request
MPI_Datatype	data type

下面舉一個簡單的例子 sample.c

```
#include <mpi.h>
void    main(int argc,    char **argv)
{
      int               nproc, myid, tag, rc;
      float             sendbuf, recvbuf;
      MPI_Request  req;
```

```
    MPI_Status     status;

    MPI_Init(&argc,    &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    tag=1;
    if (myid == 0) {
        sendbuf = 22.0;
        MPI_Send ( &sendbuf, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
    } else if (myid == 1) {
        MPI_Recv ( &recvbuf, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, status);
        printf('myid, recvbuf = %d    %f\n, myid, recvbuf);
    }
    MPI_Finalize();
}
```

上述 sample.c 在 IBM SP2 SMP 可以編譯如下 :

```
mpcc sample.c
```

其 job command file 叫作 jobc，其內容如下 :

```
#!/bin/csh
#@ network.mpi= css0,shared,us
#@ executable = /usr/bin/poe
#@ arguments = /u1/c00tch00/mpi/t6/a.out    -euilib us
#@ output       = outp2
#@ error        = outp2
#@ job_type    = parallel
#@ class        = short
#@ tasks_per_node = 2
#@ node = 1
#@ queue
```

使用 llsubmit jobc 交付執行的結果如下 :

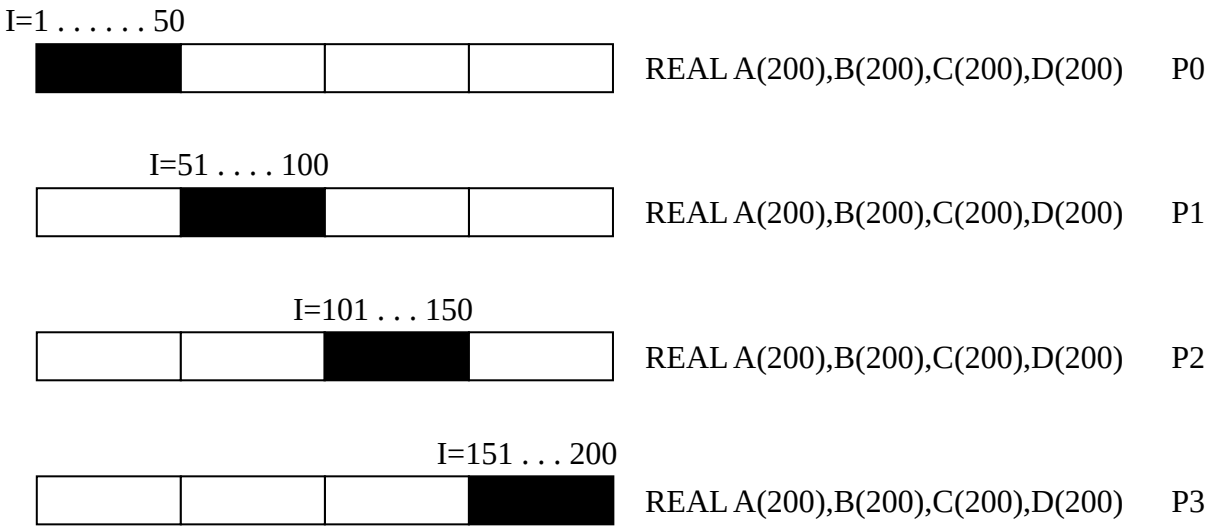
ATTENTION: 0031-408 2 tasks allocated by LoadLeveler, continuing...
myid, recvbuf = 1 22.000000

8. 多維陣列資料切割後的資料傳輸效率，由於 C 語言在記憶體上配置陣列元素時是採用 least dimension 順序，也就是最後一維 index 變動得最快而第一維變動得最慢的順序，以切割第一維的效率最好，否則就會產生不連續位址存取的現象，其傳輸效率較差。跟 Fortran 語言剛好相反。

參考書目

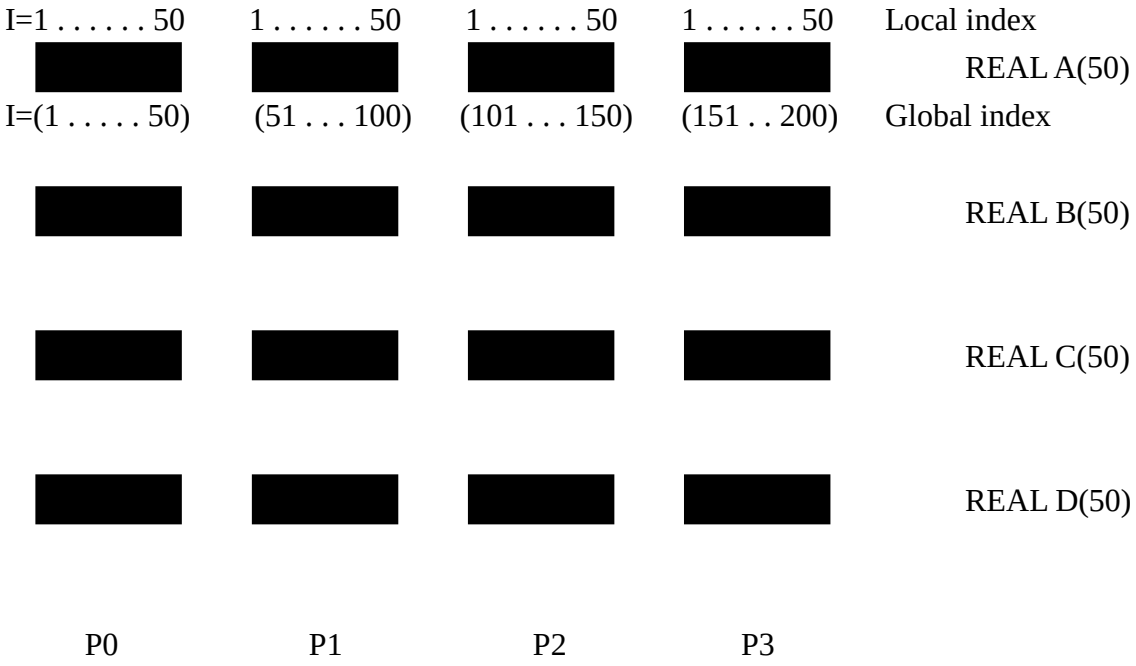
1. Tutorial on MPI : The Message-Passing Interface
   By William Gropp, Mathematics and Computer Science Division, Argonne National Laboratory, gropp@mcs.anl.gov
2. MPI in Practice
   by William Gropp, Mathematics and Computer Science Division, Argonne National Laboratory, gropp@mcs.anl.gov
3. A User’s Guide to MPI
   by Peters S. Pacheco, Department of Mathematics, University of San Francisco, peter@usfca.edu
4. Parallel Programming Using MPI
   by J.M.Chuang, Department of Mechanical Engineering, Dalhousie University, Canada chuangjm@newton.ccs.tuns.ca
5. RS/6000 SP : Practical MPI Programming
   IBM International Technical Support Organization, http://www.redbooks.ibm.com

Parallel Processing without Partition of 1-D Arrays



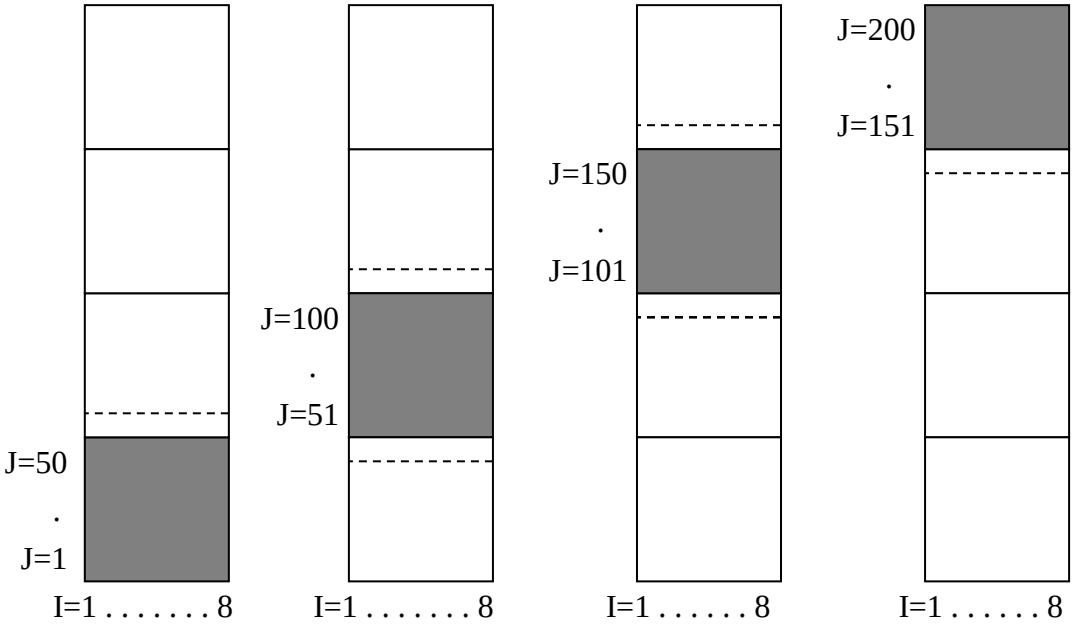

```
DO I=1,200                          DO I=ISTART, IEND
   A(I)=B(I)+C(I)*D(I)      ---->      A(I)=B(I)+C(I)*D(I)
ENDDO                               ENDDO
```

Parallel Processing with Partition of 1-D Arrays




```
DO I=1,200                          DO I=ISTART, IEND
   A(I)=B(I)+C(I)*D(I)      ---->      A(I)=B(I)+C(I)*D(I)
ENDDO                               ENDDO
```

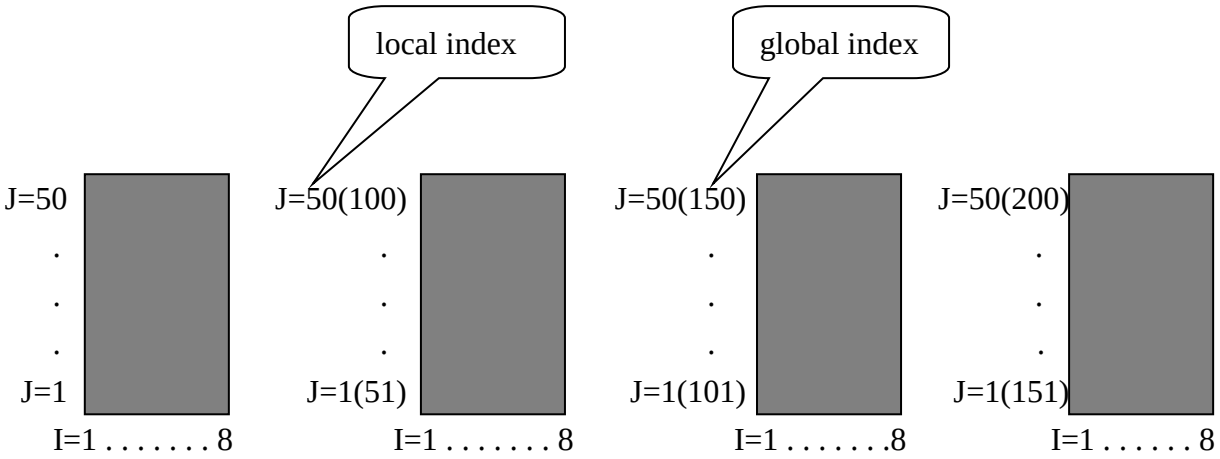
Parallel on the 2nd Dimension of 2-D Arrays without Partition



X(I,J)

P0	P1	P2	P3
X(8,200)	X(8,200)	X(8,200)	X(8,200)
Y(8,200)	Y(8,200)	Y(8,200)	Y(8,200)
Z(8,200)	Z(8,200)	Z(8,200)	Z(8,200)

Parallel on the 2nd Dimension of 2-D Arrays with Partition



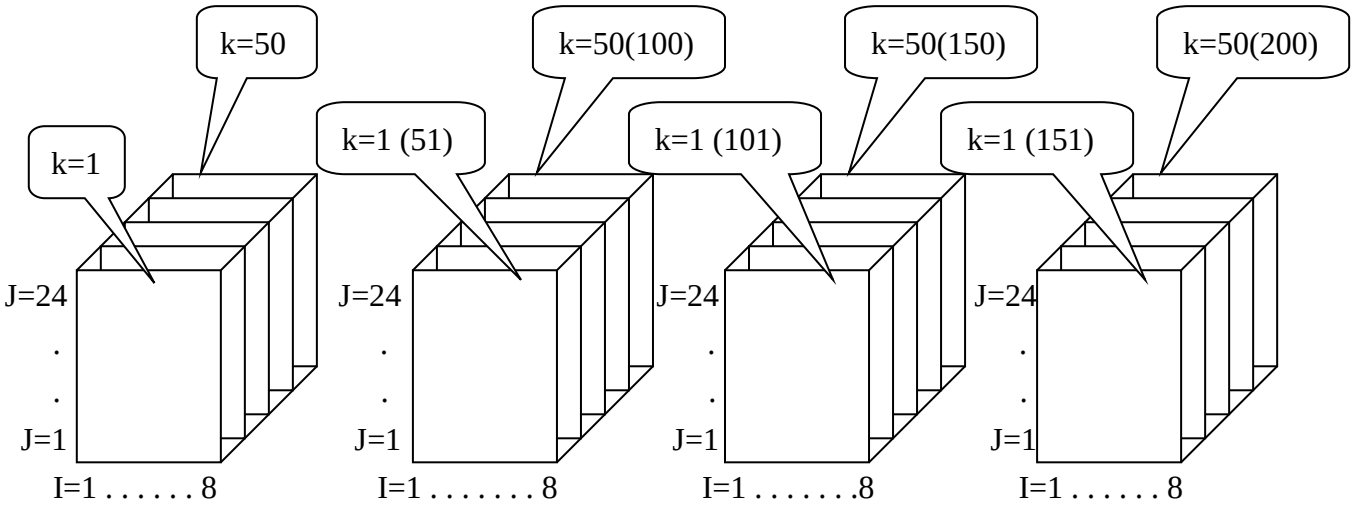
X(I,J)

X(8,50)	X(8,50)	X(8,50)	X(8,50)
Y(8,50)	Y(8,50)	Y(8,50)	Y(8,50)
Z(8,50)	Z(8,50)	Z(8,50)	Z(8,50)
P0	P1	P2	P3

Sequential Version

X(8,200), Y(8,200), Z(8,200)

Partition on the 3rd dimension of 3-D Arrays



括弧裏的數字為 Global index

X(I,J,K)

X(8,24,50)	X(8,24,50)	X(8,24,50)	X(8,24,50)
Y(8,24,50)	Y(8,24,50)	Y(8,24,50)	Y(8,24,50)
Z(8,24,50)	Z(8,24,50)	Z(8,24,50)	Z(8,24,50)
P0	P1	P2	P3

Sequential Version :

X(8,24,200), Y(8,24,200), Z(8,24,200)

```
DO J=1,N
    DO I=1,M
        Y(I,J)=0.25*(X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))+H*F(I,J)
    ENDDO
ENDDO
```

```
DO J=1,N
    DO I=1,M
        X(I,J)=0.25*(X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))+H*F(I,J)
    ENDDO
ENDDO
```

This kind of loops is called DATA RECURSIVE