Ambientes de Paso de Mensajes - Parallel Virtual Machine*

José Enrique Alvarez Estrada

Based on *Message-Passing Environments - Parallel Virtual Machine*† by

Charles Severance

Kevin Dowd

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License ‡

La idea tras PVM consiste en ensamblar una "máquina virtual" mediante un conjunto diverso de recursos enlazados por una red. Un usuario puede controlar los recursos de 35 estaciones disponibles sobre la Internet, y tener su propio sistema de procesamiento escalable. El trabajo de PVM comenzó a inicios de la década de 1990 en Oak Ridge National Labs, y en buena medida se convirtió en un éxito instantáneo en la comunidad científica. Proporciona un marco de trabajo en bruto que permite experimentar con el uso de redes de estaciones de trabajo como procesadores paralelos.

En PVM versión 3, puede usted crear su máquina virtual usando procesadores individuales, multiprocesadores de memoria compartida y multiprocesadores escalables. PVM intenta entretejer todos esos recursos en un ambiente de ejecución único y consistente.

Para ejecutar PVM, lo único que requiere usted es una cuenta de acceso en un conjunto de computadoras en red, que tengan instalado el software PVM. Incluso puede usted instalarlo en su propio directorio personal. Para crear su propia máquina virtual personal, debe integrar una lista de tales computadoras en un archivo:

```
% cat hostfile
frodo.egr.msu.edu
gollum.egr.msu.edu
mordor.egr.msu.edu
%
```

Tras ciertas manipulaciones no triviales de rutas de acceso y variables de ambiente, puede usted arrancar la consola PVM:

*Version 1.1: May 9, 2011 9:27 pm -0500
†http://cnx.org/content/m33779/1.3/
‡http://creativecommons.org/licenses/by/3.0/

```
% pvm hostfile
pvmd already running.
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
                  frodo     40000 SUN4SOL2    1000
                  gollum    40001 SUN4SOL2    1000
                  mordor    40002 SUN4SOL2    1000
pvm> ps
                   HOST     TID   FLAG 0x COMMAND
                    frodo    40042     6/c,f pvmgs
pvm> reset
pvm> ps
                   HOST     TID   FLAG 0x COMMAND
pvm>
```

Muchos usuarios distintos puede ejecutar máquinas virtuales que usan el mismo grupo de recursos. Y cada usuario a ver como una máquina vacía. La única forma de que detecte usted las otras máquinas virtuales que emplean los mismos recursos que la suya, es mediante el porcentaje de tiempo que sus aplicaciones tendrán el control de la CPU.

Existen muchos comandos que pueden ejecutarse en la consola PVM. El comando *ps* muestra los procesos en ejecución en su máquina virtual. Es factible tener más procesos que sistemas de cómputo. Cada proceso comparte el tiempo de ejecución del sistema con el resto de la carga de trabajo del mismo. El comando *reset* sirve para reiniciar su máquina virtual. Usted es el administrador de sistemas virtual de la máquina virtual que ensambló.

Para poder ejecutar programas en su máquina virtual, debe usted compilar y enlazar sus programas con las rutinas de la biblioteca PVM: [1]

```
% aimk mast slav
making in SUN4SOL2/ for SUN4SOL2
cc -O -I/opt/pvm3/include -DSYSVBFUNC -DSYSVSTR -DNOGETDTBLSIZ
      -DSYSVSIGNAL -DNOWAIT3 -DNOUNIXDOM -o mast
      ../mast.c -L/opt/pvm3/lib/SUN4SOL2 -lpvm3 -lnsl -lsocket
mv mast ~crs/pvm3/bin/SUN4SOL2
cc -O -I/opt/pvm3/include -DSYSVBFUNC -DSYSVSTR -DNOGETDTBLSIZ
      -DSYSVSIGNAL -DNOWAIT3 -DNOUNIXDOM -o slav
      ../slav.c -L/opt/pvm3/lib/SUN4SOL2 -lpvm3 -lnsl -lsocket
mv slav ~crs/pvm3/bin/SUN4SOL2
%
```

Cuando se topa con la primera llamada a PVM, la aplicación contacta con su máquina virtual y se registra a sí misma en ella. En este punto debe mostrarse como parte de la salida del comando *ps* si se ejecuta en la consola de la máquina virtual.

[1] Nota: la forma exacta de compilar puede ser diferente para su sistema.

A partir de ese punto, su aplicación realiza llamadas a PVM para crear más procesos e interactuar con ellos. PVM tiene la responsabilidad de distribuir los procesos entre los diferentes sistemas en la máquina virtual, basado en la carga y su evaluación del rendimiento relativo de cada sistema. Los mensajes se mueven a lo largo de la red usando el *user datagram protocol* (UDP), y así se entregan al proceso apropiado.

Típicamente, la aplicación PVM arranca algunos procesos PVM adicionales. Puede tratarse de copias adicionales del mismo programa, o cada proceso PVM puede ejecutar una aplicación PVM diferente. Entonces el trabajo se distribuye entre los procesos, y los resultados se reúnen cuando sea necesario.

Existen varios modelos de cómputo básicos que pueden usarse típicamente cuando se trabaja con PVM:

Maestro/Esclavo: Cuando se opera en este modo, se designa un proceso (usualmente el inicial) como el maestro, que engendra cierto número de procesos que realizarán el trabajo. Se le envían unidades de trabajo a cada uno de estos procesos, y los resultados se regresan al maestro. A menudo el proceso maestro mantiene una cola de trabajos a realizarse, de forma que cuando el esclavo finaliza, el maestro le envía a éste un nuevo elemento de trabajo. Este enfoque funciona bien cuando hay poca interacción entre datos, es decir, cuando cada unidad de trabajo es independiente, y tiene la ventaja de que el problema global balancea su carga de manera natural, incluso cuando hay variaciones en los tiempos de ejecución de los procesos individuales.

Difusión/Reunión: Este tipo de aplicación se caracteriza típicamente por el hecho de que la estructura de datos compartida es relativamente pequeña, y puede copiarse fácilmente en cada nodo de procesamiento. Al inicio de cada ciclo, todas las estructuras de datos globales se envían por difusión del proceso maestro a todos los demás. Luego cada proceso opera sobre su propia porción de los datos, y cada uno produce un resultado parcial, que es enviado de vuelta para que el proceso maestro los reúna. Este patrón se repite ciclo tras ciclo.

SPMD/Descomposición de datos: Cuando la estructura de datos es demasiado grande como para que cada proceso almacena su propia copia, ésta debe dividirse entre múltiples procesos. Generalmente, al inicio de cada ciclo, todos los procesos deben intercambiar algunos datos con cada uno de sus procesos vecinos. Después, con sus datos locales acrecentados por el necesario subconjunto de datos remotos, realizan sus cáculos. Al final del ciclo, se intercambian nuevamente los datos necesarios entre los procesos vecinos, y se reinicia el ciclo.

Las aplicaciones mas complicadas tienen flujos de datos no uniformes, así como datos que migran entre los sistemas conforme la aplicación y las cargas de trabajo cambian en el sistema.

En esta sección tenemos dos programas de ejemplo: uno es una operación maestro-esclavo, y la otra es una solución del tipo descomposición de datos, ambos para el problema del flujo de calor.

1 Cola de Tareas

En este ejemplo, un proceso (`mast`) crea cuatro procesos esclavos (`slav`) y reparte 20 unidades de trabajo (sumar uno a un número). Conforme responde un proceso esclavo, se le da un nuevo trabajo o se le dice que se han agotado todas las unidades de trabajo:

```
% cat mast.c
#include <stdio.h>
#include "pvm3.h"

#define MAXPROC 5
#define JOBS 20

main()
```

```
{
  int mytid,info;
  int tids[MAXPROC];
  int tid,input,output,answers,work;

  mytid = pvm_mytid();
  info=pvm_spawn("slav", (char**)0, 0, "", MAXPROC, tids);

/* Envía el primer trabajo */
  for(work=0;work<MAXPROC;work++) {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&work, 1, 1 ) ;
    pvm_send(tids[work],1) ;/* 1 = msgtype */
  }

/* Envía el resto de las solicitudes de trabajo */
  work = MAXPROC;
  for(answers=0; answers < JOBS ; answers++) {
    pvm_recv( -1, 2 ); /* -1 = any task 2 = msgtype */
    pvm_upkint( &tid, 1, 1 );
    pvm_upkint( &input, 1, 1 );
    pvm_upkint( &output, 1, 1 );
    printf("Gracias a %d 2*%d=%d\n",tid,input,output);
    pvm_initsend(PvmDataDefault);
    if ( work < JOBS ) {
       pvm_pkint(&work, 1, 1 ) ;
       work++;
    } else {
      input = -1;
      pvm_pkint(&input, 1, 1 ) ; /* Les indica que se detengan */
    }
    pvm_send(tid,1) ;
  }

  pvm_exit();
}
%
```

Uno de los aspectos interesantes de la interfaz PVM es la separación de llamadas para preparar un nuevo mensaje, empaquetar datos en él y enviarlo. Ello es así por varias razones. PVM tiene la capacidad de convertir entre distintos formatos de punto flotante, cambiar el orden de los bytes y traducir formatos de caracteres. Esto también permite que un único mensaje tenga múltiples elementos de datos con diferentes tipos.

El propósito del tipo de mensaje en cada envío o recepción PVM es permitir a quien envía esperar por un tipo particular de mensaje. En este ejemplo, usamos dos tipos de mensajes: el primero es un mensaje del maestro al esclavo, y el segundo es la respuesta.

Al realizar una recepción, un proceso puede o bien esperar un mensaje proveniente de un proceso específico, o un mensaje de cualquier proceso.

En la segunda fase del cómputo, el maestro espera la respuesta de cualquier esclavo, la imprime y luego reparte otra unidad de trabajo al esclavo, o le indica que termine, enviándole un mensaje con el valor -1.

El código del esclavo es muy simple: espera un mensaje, lo desempaqueta, comprueba si se trata de un mensaje de terminación, regresa una respuesta, y repite:

```
% cat slav.c
#include <stdio.h>
#include "pvm3.h"

/* Un programa sencillo para duplicar enteros */
main()
{
  int mytid;
  int input,output;
  mytid = pvm_mytid();

  while(1) {
    pvm_recv( -1, 1 ); /* -1 = cualquier tarea 1= tipo de mensaje */
    pvm_upkint(&input, 1, 1);
    if ( input == -1 ) break; /* Todo realizado */

    output = input * 2;
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &mytid, 1, 1 );
    pvm_pkint( &input, 1, 1 );
    pvm_pkint( &output, 1, 1 );
    pvm_send( pvm_parent(), 2 );
  }
  pvm_exit();
}
%
```

Cuando se ejecuta el programa maestro, produce la siguiente salida:

```
% pheat
Gracias a 262204 2*0=0
Gracias a 262205 2*1=2
Gracias a 262206 2*2=4
Gracias a 262207 2*3=6
Gracias a 262204 2*5=10
Gracias a 262205 2*6=12
Gracias a 262206 2*7=14
Gracias a 262207 2*8=16
Gracias a 262204 2*9=18
Gracias a 262205 2*10=20
Gracias a 262206 2*11=22
Gracias a 262207 2*12=24
```

```
Gracias a 262205 2*14=28
Gracias a 262207 2*16=32
Gracias a 262205 2*17=34
Gracias a 262207 2*18=36
Gracias a 262204 2*13=26
Gracias a 262205 2*19=38
Gracias a 262206 2*15=30
Gracias a 262208 2*4=8
%
```

Claramente los procesos están operando en paralelo, y el orden de ejecución es en cierta forma eleatorio. Este código es un excelente esqueleto para manejar una amplia variedad de cómputos. En el siguiente ejemplo, realizaremos un cómputo estilo SPMD para resolver el problema del flujo de calor usando PVM.

2 Flujo de Calor en PVM

El siguiente ejemplo es una aplicación mucho más complicada, que implementa el problema del flujo de calor en PVM. De muchos modos, nos da una vsión del trabajo que realiza el ambiente HPF. Resolveremos un flujo de calor en una placa bidimensional con dos fuentes de calor y los bordes inmersos en agua a cero grados, como se muestra en Figure 1 (Una placa bidimensional con cuatro fuentes constantes de calor).

Una placa bidimensional con cuatro fuentes constantes de calor

Figure 1

Los datos se repartirán entre todos los procesos usando una distribución (`*`, `BLOCK`). Las columnas se distribuyen entre procesos en bloques contiguos, y todos los elementos de los renglones en una columna se almacenan en el mismo proceso. Como con HPF, el proceso que "posee" una celda de datos realiza los cálculos para esa celda tras recibir cualquier dato necesario para realizar el cálculo.

Usamos un enfoque rojo-negro, pero por simplicidad copiamos los datos de vuelta al final de cada iteración. Para que verdaderamente fuera rojo-negro, debe usted realizar un cálculo en la dirección opuesta cada nuevo paso.

Note que en vez de engendrar procesos esclavos, el proceso padre engendra copias adicionales de sí mismo. Esto es típico de los programas estilo SPMD. Una vez engendrados tales procesos adicionales, todos los procesos esperan en una barrera antes de observar los números de proceso de los miembros del grupo. Una vez que los procesos han llegado a la barrera, todos ellos recuprean una lista de los distintos números de proceso:

```
% cat pheat.f
      PROGRAM PHEAT
      INCLUDE '../include/fpvm3.h'
      INTEGER NPROC,ROWS,COLS,TOTCOLS,OFFSET
      PARAMETER(NPROC=4,MAXTIME=200)
      PARAMETER(ROWS=200,TOTCOLS=200)
      PARAMETER(COLS=(TOTCOLS/NPROC)+3)
      REAL*8 RED(0:ROWS+1,0:COLS+1), BLACK(0:ROWS+1,0:COLS+1)
      LOGICAL IAMFIRST,IAMLAST
      INTEGER INUM,INFO,TIDS(0:NPROC-1),IERR
      INTEGER I,R,C
      INTEGER TICK,MAXTIME
      CHARACTER*30 FNAME

*     Obtener como va la cosa SPMD - Unirse al grupo pheat
      CALL PVMFJOINGROUP('pheat', INUM)

* Si somos los primeros en el grupo pheat, creamos algunos ayudantes
      IF ( INUM.EQ.0 ) THEN
        DO I=1,NPROC-1
          CALL PVMFSPAWN('pheat', 0, 'anywhere', 1, TIDS(I), IERR)
        ENDDO
      ENDIF

*     Barrera para asegurarnos que todos estamos en este punto, y así poder buscarnos
      CALL PVMFBARRIER( 'pheat', NPROC, INFO )

* Encontrar a mis camaradas y obtener sus TIDs - Los TIDS son necesarios para los envíos
      DO I=0,NPROC-1
        CALL PVMFGETTID('pheat', I, TIDS(I))
      ENDDO
```

En este punto del código, tenemos `NPROC` procesos ejecutándose en modo `SPMD`. El siguiente paso es determinar cuál subconjunto del arreglo debe calcular cada proceso, lo cual se maneja mediante la variable `INUM`, cuyo rango va de 0 a 3 y que identifica unívocamente esos procesos.

Descomponemos los datos y almacenamos sólo un cuarto de los mismos en cada proceso. Usando la variable `INUM`, elegimos nuestro conjunto continuo de columnas para almacenar y calcular. La variable `OFFSET` mapea entre una columna "global" en el arreglo completo, y una columna local en su propio subconjunto del arreglo. Figure 2 (Asignación de los elementos de la cuadrícula a los procesadores) muestra un mapa que indica cuáles procesadores almacenan cuáles elementos de datos. Los valores marcados con una B son los valores de frontera, y no cambian durante la simulación. Todos ellos se ponen a 0. Este código a menudo resulta difícil de aprehender. Realizar una distribución (`BLOCK`, `BLOCK`) requiere una descomposición bidimensional e intercambiar datos con los vecinos superior e inferior, además de los vecinos izquierdo y derecho:

Asignación de los elementos de la cuadrícula a los procesadores

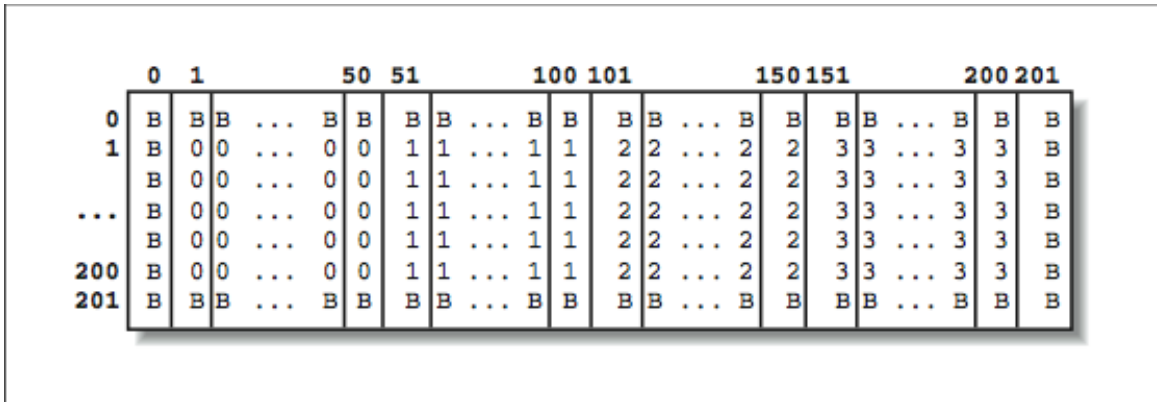

Figure 2

```
* Calcula mi geometría - ¿Qué subconjunto debo procesar? (INUM=0 valores)
* Columna actual = OFFSET + Columna (OFFSET = 0)
*     Columna 0 = vecinos a la izquierda
*     Columna 1 = enviar a la izquierda
*     Columnas 1..mylen Las celdas que me toca calcular
*     Columna mylen = Enviar a la derecha (mylen=50)
*     Column mylen+1 = Vecinos de la derecha (Columna 51)

      IAMFIRST = (INUM .EQ. 0)
      IAMLAST = (INUM .EQ. NPROC-1)
      OFFSET = (ROWS/NPROC * INUM )
      MYLEN = ROWS/NPROC
      IF ( IAMLAST ) MYLEN = TOTCOLS - OFFSET
      PRINT *,'INUM:',INUM,' Local',1,MYLEN,
     +                 ' Global',OFFSET+1,OFFSET+MYLEN

* Inicio en frío
      DO C=0,COLS+1
        DO R=0,ROWS+1
          BLACK(R,C) = 0.0
        ENDDO
      ENDDO
```

Ahora estamos ejecutando los pasos consecutivos. La primera acción en cada paso es reiniciar las fuentes de calor. En esta simulación, tenemos cuatro fuentes de calor colocadas cerca del centro de la placa. Debemos

reiniciar todos los valores cada vez que ejecutamos la simulación, conforme se ven modificados por el ciclo principal:

```
* Comenzamos la ejecución de los pasos sucesivos
      DO TICK=1,MAXTIME

* Configuramos las fuentes persistentes de calor
        CALL STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
     +     ROWS/3,TOTCOLS/3,10.0,INUM)
        CALL STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
     +     2*ROWS/3,TOTCOLS/3,20.0,INUM)
        CALL STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
     +     ROWS/3,2*TOTCOLS/3,-20.0,INUM)
        CALL STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
     +     2*ROWS/3,2*TOTCOLS/3,20.0,INUM)
```

Ahora realizamos el intercambio de los "valores fantasmas" con nuestros procesos vecinos. Por ejemplo, el Proceso 0 contiene los elementos para la columna global 50. Para calcular los valores para la columna 50 del paso siguiente , necesitamos la columna 51, que está almacenada en el Proceso 1. Similarmente, antes de que el Proceso 1 pueda calcular los nuevos valores para la columna 51, requiere los valores del Proceso 0 para la columna 50.

La Figure 3 (Patrón de comunicación de los valores fantasma) muestra cómo se transfieren los datos entre procesadores. Cada proceso envía su columna más a la izquierda al proceso a su izquierda, y su columna más a la derecha al proceso a su derecha. Dado que el primer y último procesos están rodeados por valores de frontera que no cambian a la izquierda y derecha, respectivamente, ello no es necesario para las columnas 1 y 200. Si todo se hace apropiadamente, cada proceso puede recibir sus valores fantasmas de sus vecinos izquierdo y derecho.

Patrón de comunicación de los valores fantasma

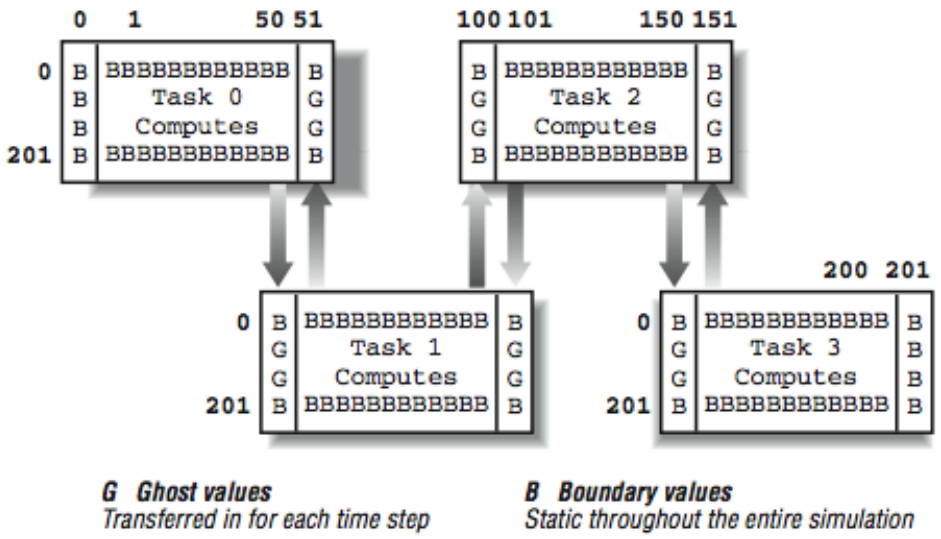

Figure 3

El resultado neto de todas las transferencias es que para cada espacio que debe calcularse, está rodeado por una capa ya sea de valores de frontera o valores fantasma de los vecinos izquierdo y derecho:

```
* Enviar izquierdo y derecho
        IF ( .NOT. IAMFIRST ) THEN
          CALL PVMFINITSEND(PVMDEFAULT,TRUE)
          CALL PVMFPACK( REAL8, BLACK(1,1), ROWS, 1, INFO )
          CALL PVMFSEND( TIDS(INUM-1), 1, INFO )
        ENDIF
        IF ( .NOT. IAMLAST ) THEN
          CALL PVMFINITSEND(PVMDEFAULT,TRUE)
          CALL PVMFPACK( REAL8, BLACK(1,MYLEN), ROWS, 1, INFO )
          CALL PVMFSEND( TIDS(INUM+1), 2, INFO )
        ENDIF
* Recibir derecho, luego izquierdo
        IF ( .NOT. IAMLAST ) THEN
           CALL PVMFRECV( TIDS(INUM+1), 1, BUFID )
           CALL PVMFUNPACK ( REAL8, BLACK(1,MYLEN+1), ROWS, 1, INFO
        ENDIF
        IF ( .NOT. IAMFIRST ) THEN
          CALL PVMFRECV( TIDS(INUM-1), 2, BUFID )
```

```
          CALL PVMFUNPACK ( REAL8, BLACK(1,0), ROWS, 1, INFO)
        ENDIF
```

El siguiente segmento es la parte fácil. Todos los valores fantasma apropiados están en su lugar, así que simplemente debemos realizar el cálculo de nuestro subespacio. Al final, copiamos de vuelta desde el arreglo `ROJO` al arreglo `NEGRO`; en una simulación real, podemos realizar dos pasos consecutivos, uno desde `NEGRO` hacia `ROJO` y el otro de `ROJO` hacia `NEGRO`, para ahorrar esta copia extra:

```
* Realiza el flujo
        DO C=1,MYLEN
          DO R=1,ROWS
            RED(R,C) = ( BLACK(R,C) +
     +                   BLACK(R,C-1) + BLACK(R-1,C) +
     +                   BLACK(R+1,C) + BLACK(R,C+1) ) / 5.0
          ENDDO
        ENDDO

* Copia de vuelta - Normalmente haríamos una versión roja y una negra de este ciclo
        DO C=1,MYLEN
          DO R=1,ROWS
            BLACK(R,C) = RED(R,C)
          ENDDO
        ENDDO
      ENDDO
```

Ahora encontramos la celda central y la enviamos al proceso maestro (si es necesario) de forma que pueda imprimirse. También volcamos los datos en archivos para depuración o visualización posterior de los resultados. Cada archivo se hace único agregando a su nombre el número de instancia. Después el programa termina:

```
      CALL SENDCELL(RED,ROWS,COLS,OFFSET,MYLEN,INUM,TIDS(0),
     +       ROWS/2,TOTCOLS/2)

* Volcado de los datos para verificación
      IF ( ROWS .LE. 20 ) THEN
        FNAME = '/tmp/pheatout.' // CHAR(ICHAR('0')+INUM)
        OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
        DO C=1,MYLEN
          WRITE(9,100)(BLACK(R,C),R=1,ROWS)
100        FORMAT(20F12.6)
        ENDDO
       CLOSE(UNIT=9)
     ENDIF
* Ponemos todo junto
      CALL PVMFBARRIER( 'pheat', NPROC, INFO )
```

```
            CALL PVMFEXIT( INFO )

            END
```

La rutina SENDCELL encuentra una celda en particular, y la imprime en el proceso maestro. Esta rutina se invoca en un estilo SPMD: todos los procesos entran en ella, si bien no necesariamente al mismo tiempo. Dependiendo del INUM y de la celda que estemos observando, cada proceso puede hacer algo diferente.

Si la celda en cuestión está en el proceso maestro, y nosotros somos el proceso maestro, imprimimos. Ninguno de los demás procesos hace algo. Si la celda en cuestión está almacenada en otro proceso, aquél que la contenga la envía al proceso maestro. El proceso maestro recibe el valor y lo imprime. El resto de los procesos no hacen nada.

Se tata de un ejemplo simple de un código con un estilo típicamente SPMD. Todos los procesos ejecutan el codigo aproximadamente al mismo tiempo, pero en base a la información local de cada proceso, las acciones realizadas por los diferentes procesos pueden ser muy distintas:

```
SUBROUTINE SENDCELL(RED,ROWS,COLS,OFFSET,MYLEN,INUM,PTID,R,C)
            INCLUDE '../include/fpvm3.h'
            INTEGER ROWS,COLS,OFFSET,MYLEN,INUM,PTID,R,C
            REAL*8 RED(0:ROWS+1,0:COLS+1)
            REAL*8 CENTER

      * Calculamos el número de renglón local, para determinar si es nuestro
            I = C - OFFSET
            IF ( I .GE. 1 .AND. I.LE. MYLEN ) THEN
              IF ( INUM .EQ. 0 ) THEN
                PRINT *,'Master has', RED(R,I), R, C, I
              ELSE
                CALL PVMFINITSEND(PVMDEFAULT,TRUE)
                CALL PVMFPACK( REAL8, RED(R,I), 1, 1, INFO )
                PRINT *, 'INUM:',INUM,' Returning',R,C,RED(R,I),I
                CALL PVMFSEND( PTID, 3, INFO )
              ENDIF
            ELSE
              IF ( INUM .EQ. 0 ) THEN
                CALL PVMFRECV( -1 , 3, BUFID )
                CALL PVMFUNPACK ( REAL8, CENTER, 1, 1, INFO)
                PRINT *, 'Master Received',R,C,CENTER
              ENDIF
            ENDIF
            RETURN
            END
```

Como la rutina previa, todos los procesos ejecutan la rutina STORE. La idea es almacenar un valor en una posición de renglón y columna *global*. Primero, debemos determinar si la celda está en nuestro proceso. Si es así, debemos calcular la columna local (I) en nuestro subconjunto de la matriz global, y luego almacenar el valor:

```
SUBROUTINE STORE(RED,ROWS,COLS,OFFSET,MYLEN,R,C,VALUE,INUM)
REAL*8 RED(0:ROWS+1,0:COLS+1)
REAL VALUE
INTEGER ROWS,COLS,OFFSET,MYLEN,R,C,I,INUM
I = C - OFFSET
IF ( I .LT. 1 .OR. I .GT. MYLEN ) RETURN
RED(R,I) = VALUE
RETURN
END
```

Cuando se ejecuta este programa, proporciona la siguiente salida:

```
% pheat
 INUM: 0 Local 1 50 Global 1 50
 El maestro recibió 100 100 3.4722390023541D-07
%
```

Vemos do líneas de impresión. La primera indica los valores que el Proceso 0 usó en su cálculo de geometría. La segunda es la salida del proceso maestro con la temperatura de la celda (100,100), tras 200 ciclos de tiempo.

Una técnica interesante, que resulta útil para depurar este tipo de programa, es cambiar el número de procesos creados. Si el programa no está moviendo sus datos apropiadamente, usualmente obtendrá usted distintos resultados cuando use distinto número de procesos. Si lo observa detalladamente, notará que el código anterior funciona correctamente de 1 a 30 procesos.

Observe que no hay una operación de barrera al final de cada ciclo de tiempo. Ello contrasta con la forma en que operan los ciclos paralelos en los multiprocesadores con acceso a uniforme a memoria compartida, que fuerzan una barrera al final de cada ciclo. Dado que hemos puesto como regla que "el dueño calcule", y que nada se calcule hasta que se hayan recibido todos los datos fantasma, no hay necesidad de tal barrera. El receptor de los mensajes con los valores fantasmas apropiados permite a un proceso comenzar a calcular de inmediato, sin importar lo que los otros procesos estén realizando en ese momento.

Este ejemplo puede usarse ya sea como un marco de trabajo para desarrollar otros cálculos basados en retículas, o como una buena excusa para usar HPF y apreciar el arduo trabajo llevado a cabo por los creadores de compiladores HPF. Una implementación bien hecha de esta simulación en HPF debe presentar mejor rendimiento que la implementación PVM, porque HPF puede llevar a cabo optimizaciones más estrictas. Al contrario que nosotros, el compilador HPF no tiene por qué hacer que sea fácilmente legible el código que genera.

3 PVM Summary

PVM es una herramienta ampliamente usada, porque proporciona portabilidad a lo largo de cualquier arquitectura diferente de la SIMD. Una vez realizado el esfuerzo de hacer que un código use paso de mensajes, tiende a ejecutarse bien en muchas arquitecturas distintas.

Las principales preocupaciones cuando se usa PVM son:

- La necesidad de una etapa de empaquetamiento, separada de la etapa de envío.

- El hecho de que esté diseñado para trabajar en un ambiente heterogéneo, puede implicar cierto nivel de sobrecarga
- No lleva cabo de manera automática tareas comunes como los cálculos geométricos

Pero a pesar de todo, para cierto conjunto de programadores, PVM es la herramienta a usarse. Si desea usted aprender más acerca de PVM vea *PVM — A User's Guide and Tutorial for Networked Parallel Computing*, de Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, y Vaidy Sunderam (MIT Press). Hay información disponible acerca de ello en www.netlib.org/pvm3/[2] .

[2] http://cnx.org/content/m38291/latest/www.netlib.org/pvm3/