El modelo PRAM

Francisco de Sande González

2.1. *Introducción*

Una *(n, M)-PRAM* consta de *n* procesadores y *M* posiciones de memoria, donde cada procesador es una máquina de acceso aleatorio [For78]. Todos los procesadores comparten la memoria y se comunican a través de ella (Figura 2.1). Durante un paso dado, cada procesador puede leer un elemento de la memoria compartida en su memoria local o escribir un elemento de su memoria local a la memoria compartida o ejecutar cualquier operación *RAM* (operaciones aritmético-lógicas) sobre un dato contenido en su memoria local. En su formulación inicial es un modelo *Multiple Instruction Multiple Data* síncrono. Ello significa que los procesadores disponen de distintos contadores de programa y pueden estar ejecutando diferentes instrucciones en un instante dado, pero todos ellos tardan el mismo número de ciclos en ejecutarlas. A pesar de ello, la mayoría de los algoritmos *PRAM* propuestos explotan el modo *Single Instruction Multiple Data* en el que los procesadores ejecutan la misma instrucción sobre diferentes datos.

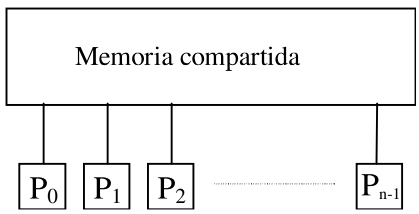

Figura 2.1 El modelo *PRAM*

2.1.1. Variantes según la resolución de los conflictos de acceso a memoria

El acceso concurrente de varios procesadores a la misma posición de la memoria compartida hace que surjan distintos modelos *PRAM*. Estas variantes son:

- La *EREW PRAM* (*Exclusive Read Exclusive Write*), donde sólo un procesador puede leer o escribir en una determinada posición de memoria a la vez.

- La *CREW PRAM* (*Concurrent Read Exclusive Write*) donde todos los procesadores pueden leer a la vez una determinada posición de memoria, pero sólo uno puede escribir en una posición de memoria en un instante.

- La *CRCW PRAM* (*Concurrent Read Concurrent Write*), donde múltiples procesadores pueden leer o escribir a la vez en una posición de memoria.

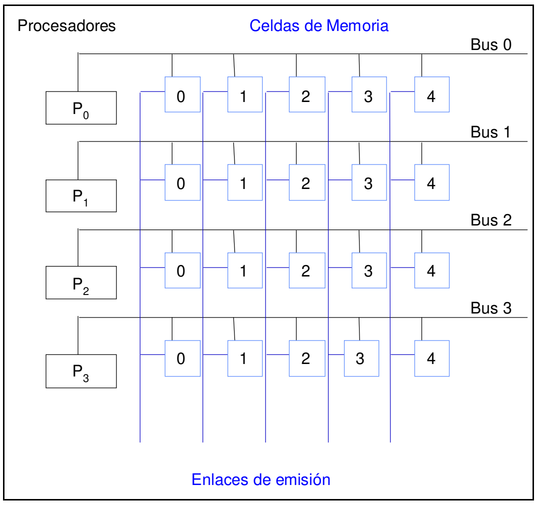

Figura 2.2 Memoria Compartida CREW de acceso constante

La *CRCW PRAM* se clasifica a su vez en cuatro variantes según su estrategia de resolución de conflicto en escritura, es decir, según qué se escribe cuando más de un procesador escribe en una misma posición de memoria en el mismo ciclo:

- *COMMON CRCW PRAM*, todos los valores escritos concurrentemente deben ser idénticos. Si no son todos iguales, se produce un error.

- *ARBITRARY CRCW PRAM*, el procesador que escribe el dato se elige arbitrariamente de entre los procesadores que escriben.

- *PRIORITY CRCW PRAM*, el procesador que escribe el dato es el procesador con mayor prioridad.

- *COMBINING CRCW PRAM*, el dato escrito es una combinación lineal de todos los valores que se escriben concurrentemente. Esta combinación se puede realizar con cualquier operación asociativa y conmutativa que sea computable en tiempo constante por una *RAM*.

2.1.2. Factibilidad del modelo PRAM

La simplicidad y generalidad del modelo *PRAM* lo han convertido en una herramienta de trabajo ampliamente aceptada, por lo que existe un gran número de algoritmos diseñados para el mismo. Sin embargo se suele cuestionar el realismo de este modelo no tanto en su calidad de modelo de programación como en su aspecto de modelo de análisis de rendimiento. Para que la memoria compartida pueda ser accedida en tiempo constante debe existir un camino físico desde el procesador hasta la posición de memoria. Se necesitarán puertas lógicas que decodifiquen la dirección. Si la memoria tiene *M* posiciones, el coste de los circuitos de decodificación se expresaría como *f(M)*, siendo *f* alguna función de coste. Si *P* procesadores comparten esa memoria, el coste sube a *P*f(M)* lo que para valores de *M* y *P* grandes será excesivo. En la Figura 2.2 se muestra una posible organización de una memoria compartida CREW con costo *O(M*P),* que fue propuesta en [Leo97]. Cada dirección de memoria *d* se replica en cada procesador. La columna de celdas etiquetada *d* representa la misma dirección de memoria *d*. Cuando un procesador P_i realiza un acceso de lectura simplemente lee de su réplica correspondiente usando su bus privado Bus_i. Cuando un valor es almacenado por el procesador P_i usando Bus_i en su réplica de la dirección compartida *d*, la celda receptora transmite inmediatamente el nuevo valor a las otras celdas replicantes usando el enlace de emisión vertical *d* correspondiente.

La suposición de que los accesos a memoria compartida ocurren en tiempo constante conduce en ocasiones a la concepción de algoritmos cuya eficiencia puede ser pobre en las máquinas paralelas actuales, especialmente si la labor de adaptación del algoritmo PRAM es dejada totalmente en manos de un compilador.

2.2. Activación y asignación de procesadores

Los algoritmos publicados para el modelo PRAM se describen utilizando un estilo general y algo difuso que, se considera, no necesita una definición formal para un lector con experiencia en el campo de la programación secuencial. Ejemplos de este estilo común pueden encontrarse en los libros de Quinn [Qui90] y de Gibbons y Rytter [Gib88]. La expresión del paralelismo se hace mediante una sentencia de la forma:

for all x **in** m..n **in parallel do** instruction(x)

La semántica de la sentencia consiste en:

(a) Asignar un procesador a cada elemento *x* en *{m,…,n}*.

(b) Ejecutar, en paralelo y para cada procesador asignado, todas las operaciones especificadas en *instruction(x)*.

La ejecución de la sentencia paralela termina cuando todos los procesadores implicados completan sus computaciones individuales.

Si se asume un coste logarítmico en el número de procesadores solicitados, entonces la complejidad del código en la Figura 2.3 será *O(K log (N))*. Sin embargo, y aunque los textos suelen omitir la explicación de esta cuestión, se suele cargar el código de la Figura 2.3 con una *complejidad O(K)*.

```
for i := 1 to K do
begin
  ...
  for all x in 1..N in parallel do
    instruction(x);
  ...
end;
```

Figura 2.3 Bucle secuencial con una sentencia *for all*

```
...
spawn(N);
...
for i := 1 to K do
begin
  ...
    for all x in a..b in parallel do
      instruction(x);
  ...
end;
```

Figura 2.4 Sentencias *spawn* y *for all*

Es esencial diferenciar entre los conceptos de sentencia de "activación de procesadores" y sentencia de "asignación de procesadores". En la presentación de algoritmos PRAM a menudo la activación de procesadores es denotada de la siguiente forma:

spawn(<Total Number Of Processors>)

mientras que la construcción **for all** anterior es utilizada para denotar la sentencia de asignación de procesadores. Un tiempo de arranque de orden *log(P)* es necesario para activar *P* procesadores. La idea es que el primer procesador activa al segundo y, en el paso general, el número de procesadores activos es doblado haciendo que cada uno de los procesadores activos active a uno diferente.

Puesto que al comienzo hay un sólo procesador activo, la ejecución de un programa PRAM tiene dos fases: en la primera es activado el número total de procesadores requeridos y, en la segunda, estos procesadores realizan la computación paralela ejecutando las sentencias de asignación interiores en tiempo constante (Figura 2.4).

Explicaremos cómo puede realizarse una sentencia de asignación de procesadores en tiempo constante. Consideremos que al comienzo de la computación disponemos de M procesadores que han sido proporcionados en tiempo $O(\log M)$ por la llamada a la sentencia *spawn(M)*. Suponemos la existencia de un número "físico" de índice f, con f en $H = \{0,...,M-1\}$ asociado con cada procesador activo. El conjunto H de procesadores replica la ejecución de la parte secuencial del programa hasta que se alcanza una primera sentencia paralela de la forma:

for all *x* **in** *a..b* **in parallel do**
instruction(*x*)

El conjunto H de los M procesadores es particionado en $r = b-a+1 \leq M$ conjuntos $\{H_a,..., H_b\}$.

Cada procesador, utilizando su número físico f decide en términos de una función que determina la **política de asignación** $Map(f) = x \in \{a,...,b\}$ a qué conjunto $H_{Map(f)} = H_x = \{f \in H \,/\, Map(f) = x\}$ se incorporará. La aplicación *Map* debe ser sobreyectiva, esto es, todo proceso debe tener asignado un procesador. La familia de conjuntos H_x constituye una partición de H:

$$H = \cup_{x\in\{a,....,b\}} H_x \;;\; H_x \cap H_y = \varnothing \text{ si } x \neq y; \;\; H_x \neq \varnothing \quad \forall x\in\{a,..,b\}$$

Cada procesador f asigna a su variable x el valor $Map(f)$ entre a y b, cambia su valor de M a $M = |\, H_{Map(f)} \,|$ y su nombre físico de acuerdo con una función F de asignación de nombres, $f = F(f)$ de tal manera que los nuevos nombres físicos de los procesadores del grupo H_x pertenecen al conjunto $\{0,...,|H_x|-1\}$. Para que F sea una numeración correcta es necesario que cumpla la condición de ser inyectiva dentro de cada conjunto H_x:

$$\forall x \in \{a,..,b\},\ \forall f,\ g \in H_x\ /\ f \neq g \Rightarrow F(f) \neq F(g)$$

Desde ese momento, los procesadores pertenecientes al conjunto H_x están todos en el mismo estado, tienen el mismo valor de la variable x y replican la misma ejecución paralela. Mientras que lo común es que un procesador soporte varias tareas (virtualización del procesador en varios procesos, como se explica en [Leo95a]), en esta etapa inicial en la que existen más procesadores que tareas, una tarea es soportada por varios procesadores (replicación de la tarea en los procesadores) [Rod98d].

$$\begin{aligned} &Map: H \rightarrow \{a,...,b\} \\ &\qquad f \rightarrow Map(f) = x \\ \\ &F: H \rightarrow \cup_{x\in\{a,...,b\}} \{0,...,|H_x|-1\} \\ &Si\ f \in H_x \ \rightarrow F(f) \in \{0,...,|H_x|-1\} \end{aligned}$$

Figura 2.5 La pareja (*Map, F*) define una política de asignación de procesadores

Una política factible para las funciones de asignación de tareas *Map* y de nombres físicos *F* es la de distribución cíclica:

$$s = M \bmod r;\ k = f \bmod r;$$

$$\text{(Ec. 2.1)} \qquad Map(f) = a + k = x;\ \ F(f) = f \operatorname{div} r;$$

$$M = M/r \text{ si } k \geq s \text{ y } M = M/r + 1 \text{ si } k < s$$

El tiempo invertido en la ejecución de esta política cíclica de la sentencia de asignación paralela es constante. Todas las evaluaciones únicamente implican variables privadas, sin que haya requerimientos de comunicación o accesos a memoria compartida. Cuando una nueva sentencia de asignación de procesadores es ejecutada por los procesadores en el conjunto H_x:

for all *y* **in** *c[x]..d[x]* **in parallel do**
instruction(*y*)

el conjunto H_x es dividido en tiempo constante en $r_x = d[x]-c[x]+1$ subconjuntos: $\{H_{x_1},..., H_{x_{r_x}}\}$. El procesador f se incorpora al grupo $H_{Map(f)}$, asigna su identificador $y = Map(f)$ y cambia sus valores de $f = F(f)$ y $M = |H_{Map(f)}|$.

Siguiendo la política cíclica explicada anteriormente:

$$s = M \bmod r_x;\ k = f \bmod r_x;$$

$$\text{(Ec. 2.2)} \qquad Map(f) = c[k] + k = y;\ \ F(f) = f \operatorname{div} r_x;$$

$$M = M / r_x \text{ si } k \geq s \ \text{ y } \ M = M / r_x + 1 \text{ si } k < s$$

Este algoritmo muestra que la política de asignación de procesadores cíclica sólo requiere tiempo constante. Obviamente, el número de políticas de asignación a seguir es innumerable. Entre las alternativas a utilizar se encuentran las políticas de distribución por bloques y las mixtas por bloques y cíclicas. Una política de asignación de procesadores **óptima** es aquella que distribuye los procesadores de acuerdo con la distribución de procesadores que requerirá la ejecución de *instrucción(x)*. Estudiaremos este tema más adelante en este mismo Capítulo.

2.2.1. Ejemplo

```
 1   ...
 2  spawn(6);
 3  ...
 4  for all x in 3..5 in parallel do
 5  begin
 6    ...
 7      for all y in 8..9 in parallel do
 8        begin
 9           ...
10        end;
11   ...
12  end;
```

Figura 2.6 Ejemplo con bucles *for all* anidados

Consideremos el código de la Figura 2.6.

En caso de éxito la llamada a *spawn* en la línea 2 activará seis procesadores físicos. Cada uno de estos procesadores tendrá sus valores de las variables $M = |H|$ y f con valores iniciales:

$M = \|H\| =$	6	6	6	6	6	6
$F(f) = f =$	0	1	2	3	4	5

La sentencia paralela de la línea 4 solicita $r=5-3+1=3$ procesadores. De acuerdo con las fórmulas de la asignación cíclica, los nuevos valores son:

$M = \|H\| =$	2	2	2	2	2	2
$F(f) = f =$	0	0	0	1	1	1
$Map(f) = x =$	3	4	5	3	4	5

La sentencia de asignación paralela anidada en la línea 7 demanda $r=9-8+1=2$ procesadores. Los nuevos valores para $M = |H|$, f e y computados siguiendo las fórmulas 0 son:

$M = \|H\| =$	1	1	1	1	1	1
$F(f) = f =$	0	0	0	0	0	0
$Map(f) = x =$	3	4	5	3	4	5
$y =$	8	8	8	9	9	9

```
C:\>ppc rbatcher.ll
rbatcher.eje generated
rbatcher.lst generated
C:\>_
```

Figura 2.7 Ejemplo de ejecución del compilador de *La Laguna*

2.3. *El lenguaje La Laguna (ll)*

El lenguaje *La Laguna* (*ll)* [Leo97], es un lenguaje orientado al modelo PRAM. Está basado en Pascal- [Han85], por lo que su aprendizaje es inmediato para cualquiera familiarizado con el lenguaje Pascal. Desde 1990 no han dejado de aparecer nuevas versiones [Leo97], [Leo95a], [San96]. Existen versiones para diversas plataformas, tanto secuenciales como paralelas. El software y los manuales de *ll* pueden encontrarse en la referencia [Pra]. La capacidad de *ll* para expresar algoritmos PRAM es algo que ya hemos probado extensamente durante el tiempo de vida del lenguaje, tanto mediante su uso en el entorno académico (se han realizado diferentes proyectos de Ingeniería en Informática basados en el lenguaje) como a través de las diferentes publicaciones que al respecto hemos realizado.

```
program BatcherBitonicSort;
const  maxElements = 8191;  logmaxElements = 13;
type   vector = array [0..maxElements] of integer;
var  x: shared vector;   n ,  logn: shared integer;

procedure readVector;
var i: integer;
begin  readln(n); readln(logn);  for i := 0 to n-1 do read(x[i]); end;

procedure writeVector;
var i: integer;
begin   for i := 0 to n-1 do  write('x[',i,'] = ',x[i],' ');
end;

shared procedure bSort(left, right: shared integer;
                       direction: shared boolean);
begin
  if right > left then
  var center: shared integer;
  begin
     center := (left + right) div 2;
     parallel left..center do
     var i, temp1, temp2, xname, xi: integer;
     begin
         i := center+name-left+1; xname := x[name] ;  xi := x[i];
         temp1 := xname min xi;  temp2 := xname max xi;
         if direction then begin x[name] := temp1; x[i] := temp2; end
         else   begin x[name] := temp2; x[i] := temp1; end
     end;
     parallel do  bSort(left,center,direction) ||
                  bSort(center+1,right,direction);
  end
end (* bSort *);

procedure bitonicSort;
var   length, i, numOfIntervals: integer;
begin
   for i := 1 to logn do
   begin
      length := 1 shl i; numOfIntervals := n div length;
      parallel 0..numOfIntervals-1 do
      var first, last: integer;
      begin
         first := name*length;  last := first+length-1;
         bSort(first, last, (name mod 2 = 0));
      end;
   end;
end;
begin  readVector;   bitonicSort; writeVector; end.
```
Figura 2.8 Ejemplo rbatcher.ll

Considere el fichero fuente *rbatcher.ll* que se muestra en la Figura 2.8. La asignación de procesadores en *ll* se hace a través de una sentencia de la forma:

parallel <Expression1> .. <Expression2> do <Statement>

Se crean *<Expression2>-<Expression1>+1* procesos con identificadores de proceso entre *<Expression1>* y *<Expression2>*. La palabra reservada *name* denota el nombre lógico del proceso. Una segunda forma de asignar procesadores, utilizada en el procedimiento *bSort*, es a través de la sentencia de llamada paralela.

parallel do <procedure call> || ...|| *<procedure call>*

El cualificador *shared* es utilizado para indicar que una variable es compartida. Una variable compartida es aquella que será utilizada (accedida para lectura o escritura) no sólo por el proceso que le asigna espacio, sino por alguno de los procesos que fueron creados mediante sentencias de asignación de procesadores que fueron ejecutadas con posterioridad a la creación de la variable compartida. Cuando el cualificador *shared* se aplica a un procedimiento indica que ese procedimiento será invocado desde el ámbito de una sentencia paralela (por tanto su código debe ser compartido por varios procesadores). Por el contrario, una variable privada sólo puede ser utilizada por el proceso que la crea. Por defecto, en *La Laguna* las variables son privadas. *La Laguna* extiende el conjunto de operadores binarios de *Pascal.* Los operadores binarios *min* y *max* usados en el procedimiento *bSort* de la Figura 2.8 devuelven el mínimo y el máximo de los operandos. El código de la Figura 2.8 ordena el vector *x* utilizando el algoritmo de ordenación bitónica de Batcher [Bat82] que explicaremos más adelante en este Capítulo. En general, la semántica del lenguaje es la habitualmente utilizada en la mayoría de los artículos que utilizan el modelo PRAM. La Figura 2.7 muestra una compilación desde la línea de comandos.

```
PRAM Ver. 97.1
PRAM|T0>load 0, rbatcher
Load done.
P:    0 Name:   0    B:      0  S:     0: | PC:     0
G: 16 BT:   1023  SB:     0 SS:     0: | PROGRAM 1 2 51 1 1
PRAM|T0>go
8
3
4 5 2 7 8 9 1 6
x[0]=1 x[1]=2 x[2]=4 x[3]=5 x[4]=6 x[5]=7 x[6]=8 x[7]=9
PRAM|T1225>quit
```

Figura 2.9 Ejecución con el intérprete PRAM

El compilador de *ll* produce como salida un fichero *rbatcher.eje* y un fichero *rbatcher.lst*. El fichero *.eje* contiene el código intermedio ejecutable mediante el intérprete *pram*. El fichero *.lst* contiene el listado del código fuente y el correspondiente código ensamblador. La ejecución del fichero "*rbatcher.eje*" la realizaremos mediante una llamada al intérprete *pram* desde la línea de comandos. Es opcional especificar el nombre del programa a ejecutar en la línea de comandos:

*X>**pram rbatcher.eje***

Si no se especifica el nombre del fichero,

*X>**pram***

se puede cargar posteriormente mediante el comando *load addr, fileName*. Este comando tiene dos argumentos, en el primero se especifica la dirección y en el segundo el nombre del fichero.

En el ejemplo que nos ocupa, el comando *load 0, rbatcher* carga el ejecutable a partir de la posición 0 de memoria. Después de la carga del programa, el comando *load* muestra el estado del procesador con nombre físico 0 en el grupo inicial (Figura 2.9). Con el comando *go* se puede ejecutar el programa.

2.3.1. Implementaciones

Como se ha comentado, existen implementaciones del lenguaje para diferentes plataformas. En [San93] se puede encontrar un estudio detallado de la primera implementación paralela del lenguaje, que se realizó para redes de transputers [Car91] empleando occam [Inm88] como lenguaje de programación. En [Bar97] se implementa el compilador utilizando lex y yacc y se diseña también un intérprete secuencial del código producido por el compilador. En [Lun98] se desarrollaron intérpretes paralelos para el compilador utilizando PVM, MPI e Inmos C [Inm90]. Las últimas implementaciones del lenguaje [Lun98] tienen en cuenta las ideas del modelo de computación colectiva que presentaremos en el Capítulo 4. En estas versiones del lenguaje, el código *ll* es traducido por el compilador a código en C que es compilado a código paralelo con el apoyo de librerías para la ejecución paralela que han sido diseñadas utilizando paso de mensajes explícito con PVM, MPI o Inmos C.

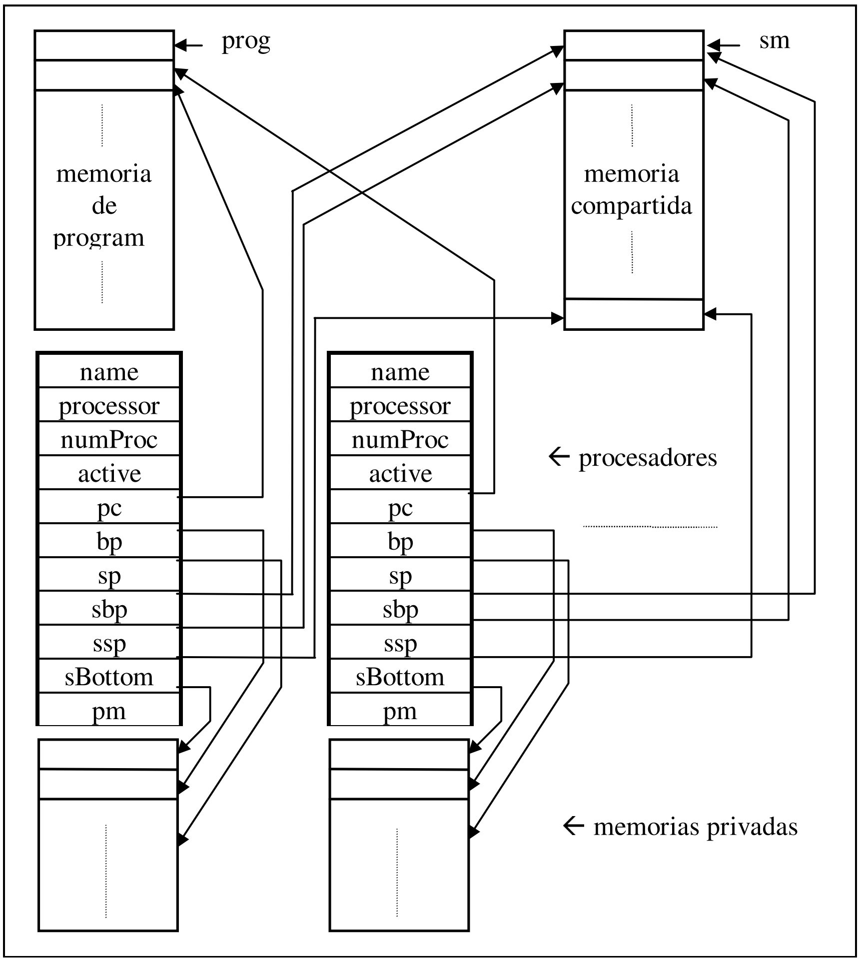

Figura 2.10 Arquitectura de la máquina objeto del compilador de *ll*

La Figura 2.10 es un esquema de la arquitectura genérica para la que el compilador de *ll* genera código. En las versiones interpretadas, el compilador genera código orientado a pila para esta máquina genérica. La máquina objeto del compilador consta de un número de procesadores que se puede cambiar a través de un fichero de configuración, una memoria de programa común a todos ellos (todos los procesadores ejecutan el mismo programa), una memoria compartida y una memoria privada para cada procesador que son manejadas en forma de pila. Cada procesador está caracterizado por el conjunto de registros que aparecen en la figura, entre los que destacamos:

- *name:* número lógico del procesador en el programa;
- *numProc:* número de procesadores por grupo;
- *active*: flag indicador de si el procesador se encuentra activo o inactivo;
- *pc*: contador de programa;
- *bp, sp: puntero de base* y *puntero de pila* para acceder a la memoria privada del procesador;

```
parallel <Expr1> .. <Expr2> do
  parallel <Expr3.. <Expr4> do
    begin
    S;
    end;

valor1 := Evaluación(Expr1)
Push(valor1)
valor2 := Evaluación(Expr2)
Push(valor2)
PATTERN
valor3 := Evaluación(Expr3)
Push(valor3)
valor4 := Evaluación(Expr4)
Push(valor4)
PATTERN
Código(S)
ENDPATTERN
ENDPATTERN
```

Figura 2.11 La traducción de sentencias paralelas anidadas

- *sbp, ssp, sBottom*: puntero de base, de pila y fondo compartidos para el acceso al segmento de la memoria compartida correspondiente a cada procesador.
- *pm*: memoria privada para cada procesador.

Las primeras versiones del compilador utilizaban un analizador sintáctico descendente recursivo, mientras que las últimas han sido escritas utilizando lex y yacc. El compilador reconoce una gramática muy similar a la de Pascal, que ha sido ampliada para soportar las sentencias paralelas, sentencias de semibloque, así como la declaración de variables y procedimientos compartidos entre otros elementos específicos.

En las versiones interpretadas del lenguaje (ya sea sobre máquinas secuenciales o paralelas) el compilador produce instrucciones orientadas a pila de la forma

$$OPCODE\ arg_1\ arg_2 \ldots arg_N$$

la mayoría de estas instrucciones son ejecutadas por cada procesador de la máquina de forma individual, sin precisar la cooperación con otros. Así por ejemplo la instrucción ADD toma dos operandos de la pila (privada) del procesador, y los suma almacenando el resultado también en la pila. Las instrucciones PATTERN o SHASSIGN que se emiten como resultado de la traducción de una sentencia *parallel*, o una escritura a memoria compartida respectivamente, son ejemplos de instrucciones que sí precisan de comunicación entre los procesadores para su ejecución.

La Figura 2.11 muestra el esquema de traducción que el compilador utiliza para traducir las sentencias *parallel*. La instrucción PATTERN es la que cambia el estado paralelo de un procesador. Como consecuencia de la ejecución de la primera sentencia *parallel* (linea 1 en la Figura 2.11) el conjunto de procesadores de la máquina se particiona en *valor2-valor1+1* subconjuntos siguiendo una determinada política. La Figura 2.12 presenta un pseudocódigo de la implementación de la política cíclica que ya hemos presentado anteriormente.

```
1 Pop(A)
 2 Pop(B)
 3 n = B - A + 1
 4 IF n > NUMPROC THEN
 5    RunTimeError()
 6 IF (ACTIVE) THEN
 7   BEGIN
 8   S := NUMPROC mod n
 9   aux := PROCESSOR mod n
10   NAME := A + aux
11   PROCESSOR := PROCESSOR div n
12   NUMPROC := NUMPROC div n
13   IF (aux < S) THEN
14     NUMPROC := NUMPROC + 1
15   END
```

Figura 2.12 La implementación de la política cíclica de distribución de procesadores

En las primeras implementaciones del lenguaje se virtualizaban procesadores: un mismo procesador físico se encargaba de la simulación de un cierto número de procesadores virtuales o lógicos [Leo95a], [Leo96a] pero debido a la ineficiencia que esta simulación por software introduce, las últimas versiones producen un error si el número de procesadores demandados es superior al número de procesadores físicos disponibles, como vemos en el código de la figura (líneas 4 y 5).

En las implementaciones paralelas del lenguaje siempre hemos tomado la aproximación de una memoria compartida que está totalmente replicada en los procesadores de la máquina. Un acceso de lectura a memoria compartida queda así convertido en un simple acceso a la memoria local del procesador, mientras que un acceso para escritura a la memoria compartida ha de traducirse en el envío de mensajes para actualizar la copia de la dirección modificada que reside en otros procesadores.

Una sentencia condicional o una sentencia de repetición con una condición que es privada (que puede tener diferente valor en diferentes procesadores) provoca que haya procesadores que ejecuten el cuerpo de la sentencia, mientras que otros no lo ejecutarán. El registro *active* de los procesadores almacena este estado del procesador. Para mantener la sincronización PRAM entre los procesadores, si hay algún procesador que haya de ejecutar una parte del código, todos los procesadores de la máquina la ejecutarán, con la diferencia que unos lo harán como procesadores *activos* y otros como procesadores *inactivos*. De este modo, cuando se produce una escritura en memoria compartida todos los procesadores envían el mensaje correspondiente al valor que han escrito y todos reciben mensajes del resto de procesadores. Sólo aquellos procesadores que tienen activo su flag *active* realizarán actualizaciones de la memoria compartida. Con este modo de trabajo se garantiza en todo momento la coherencia de la memoria compartida.

```
 1 void replicateSHmem(int ACTIVE, int len, int addr, int ptr) {
 2   int i, j, k;          /* Contadores. */
 3   int **data;           /* informacion recibida de cada procesador. */
 4   int msglen;           /* Longitud de los mensajes. */
 5   int nummsg;           /* # de mensajes a enviar */
 6   int vecino;
 7   int atid, atag, alen;
 8   int a;                /* Direccion donde se actualizará la memoria. */
 9
10   msglen = len + 2;
11   /* Memoria para los datos a enviar y recibir */
12   data = (int **)malloc(NUMPROCESSOR*sizeof(int *));
13   for (i = 0; i < NUMPROCESSOR; i++) data[i] = (int *)malloc(msglen*sizeof(int));
14
15    /* Inicializar los datos del procesador */
16   data[0][0] = ACTIVE;  data[0][1] = addr;
17   for (i = 2; i < msglen; i++)
18     data[0][i] = reg.pm[ptr + i - 2];
19
20   /* Envío y recepción */
21   for (i = 0; i < dim; i++) {
22     nummsg = 1 << i;
23     vecino = reg.PhysicName ^ (1 << i);
24     for (j = 0, k = nummsg; j < nummsg; j++, k++) {
25       pvm_initsend(ENCODING);
26       pvm_psend(tids[vecino], MSG_REPL, (char *)data[j], msglen, PVM_INT);
27       pvm_precv(tids[vecino], MSG_REPL, (char *)data[k], msglen, PVM_INT, &atid,
28       atag, &alen);
30     }
31   }
32   /* Actualización de la memoria */
33   for (i = 0; i < NUMPROCESSOR; i++)
34     for (j = 0; j < len; j++)
35       if (data[i][0] == TRUE) {        /* Msg valido */
36         a = data[i][1];                /* addr */
37         sm[j + a] = data[i][j + 2];    /* Actualizacion */
38       }
39   for (i = 0; i < NUMPROCESSOR; i++)
40     free(data[i]);
41     free(data);
42 }
```

Figura 2.13 Replicación de la memoria compartida en PVM

El código de la Figura 2.13 presenta la implementación utilizando PVM de la rutina *replicateSHmem()* que ejecutan todos los procesadores como consecuencia de una escritura en memoria compartida. Entre las líneas 20 y 31 se producen las comunicaciones y el bucle de la línea 33 realiza las actualizaciones de la memoria compartida supuesto que el mensaje es válido (ha sido enviado por un procesador *activo*.

2.3.2. Ejemplos

2.3.2.1. Suma de prefijos

Dado un vector inicial *Mk* de tamaño *n*, el problema de la suma de prefijos consiste en obtener las sumas parciales *sum[j]* definidas según la fórmula:

```
const MAXN = ...;
type VectInt = array [1..MAXN] of integer;
var   n: shared integer;
      S, sum, Mk: shared VectInt;
```

Figura 2.14 Declaraciones

$$sum[j] = \sum_{i=1,j} Mk[i] \quad j \in \{1,...,n\}$$

Supongamos las declaraciones de la Figura 2.14.

El problema puede ser resuelto en paralelo teniendo en cuenta que para hacer la suma de prefijos de un intervalo *sum[i...j]* podemos hacer en paralelo la suma de prefijos de la primera mitad *sum[i..(i+j)/2]* y de la segunda mitad *sum[(i+j)/2+1..j]*. Luego sólo tendremos que sumar el total de los elementos de la primera mitad a cada una de las sumas de prefijos de la segunda mitad. Esto último puede hacerse también en paralelo. El código *ll* de la Figura 2.16 es la codificación directa de esta idea. La Figura 2.15 es el resultado de la ejecución del algoritmo sobre una red de cuatro estaciones de trabajo. La versión del intérprete utilizada es la desarrollada en PVM.

```
shared procedure prefix(i,j: shared integer);
var k: shared integer;
begin
  if i < j then begin
    parallel 1..2 do
    var a, b: integer;
    begin
      if name = 1 then begin a := i; b := (i+j) div 2; end
      else  begin  a := ((i+j)div 2)+1; b := j;   end;
      prefix(a, b);
    end;
    k := (i+j) div 2;
    parallel k + 1..j do  sum[name] := sum[name]+sum[k];
  end
  else sum[i] := Mk[i] (* only one item in i..j *)
end; { prefix }
```

Figura 2.16 Suma de prefijos

```
ll Interpreter Ver 1.0   12.Feb.98 (c) PCGULL 1998
Version for LAN Architecture.

Reading config file...
        Number of processors:       4
        Hipercube dimension:        2
        Program size:            4096 bytes.
        Private memory size:     4096 bytes.
        Shared memory size:      8192 bytes.
Loading program...
Sending program to the nodes...

Executing...
introduzca el array:
a[4]=2
a[5]=5
a[6]=1
a[7]=3
resultados:
b[4]=2
b[5]=7
b[6]=8
b[7]=11
```

Figura 2.15 Resultado de la ejecución de la suma de prefijos

2.3.2.2. Ordenación bitónica

El algoritmo de ordenación bitónica fue propuesto por Batcher en [Bat82]. Una secuencia bitónica *x* de longitud *n* es aquella que, al considerarla como un ciclo (esto es, el siguiente del último elemento es el primero) se puede descomponer en una subsecuencia no decreciente seguida de una secuencia no creciente:

$$\exists\, i, j \in \{0, \ldots, n-1\} \,/\, x_j \leq x_{j+1} \leq \ldots \leq x_i \;\; y \; x_i \geq x_{i+1} \geq \ldots \geq x_{j-1}$$

donde todas las operaciones en los índices se realizan en aritmética módulo *n*.

El procedimiento *bSort* de la Figura 2.8 explota la siguiente propiedad de las secuencias bitónicas:

Si la secuencia $x = (x_0, \ldots, x_n)$ es bitónica, entonces las subsecuencias $y=(y_0,\ldots,y_{n/2-1})$ y $z = (z_0,\ldots,z_{n/2-1})$ dadas por:

$$y_i = min\ \{x_i, x_{i+n/2}\};\;\; z_i = max\ \{x_i, x_{i+n/2}\} \text{ con } i \in \{0,..n/2-1\}$$

son también bitónicas y ocurre que

$$y_j \leq z_i \;\; \forall\, i, j \in \{0,..n/2-1\}$$

Esta propiedad es consecuencia de que la secuencia de las y_i tiene a lo sumo un único punto de corte con la secuencia de las z_i. El procedimiento *bitonicSort* ordena sucesivamente intervalos de longitud creciente usando *bSort*. Puesto que cualquier secuencia de longitud dos es bitónica, puede ser ordenada mediante una llamada a *bSort*. El intervalo total es particionado en secuencias de longitud dos que se ordenan en paralelo. Las secuencias pares se ordenan de manera creciente y las impares de manera decreciente. Ahora los subintervalos de longitud cuatro son bitónicos y se les puede ordenar usando *bSort*. El resultado será que los subintervalos de longitud ocho son bitónicos. Después de *log(n)* iteraciones repitiendo este procedimiento el vector *x* está ordenado.

2.3.2.3. Un Quicksort paralelo

El siguiente programa *ll* constituye un ejemplo paradigmático de los requerimientos de los algoritmos desarrollados para el modelo de programación PRAM. Utiliza tres procedimientos paralelos que se anidan en sucesivas llamadas, hace un uso intenso de la memoria compartida y de la capacidad de sincronización que ofrece el modelo. El algoritmo, propuesto en [Jaj92], paraleliza las llamadas recursivas del conocido procedimiento Quicksort [Hoa61], también paraleliza el procedimiento de partición y para ello hace a su vez uso de un procedimiento paralelo para la suma de prefijos. Después de leerlo es aconsejable preguntarse si un algoritmo como este puede ser implementado eficientemente en un computador distribuido.

Sea *S* un vector que contiene los *n* elementos a ordenar (declaraciones de la Figura 2.14) y denotemos por *lft* y *rgt* los extremos izquierdo y derecho del vector a ordenar. Se toma aleatoriamente un elemento pivote *S[pp]* del vector de entrada *S*. La estrategia del Quicksort es la de reorganizar utilizando un procedimiento de división *part(lft, rgt, pp)* los elementos de *S* en dos subvectores *S[lft..pp-1]* y *S[pp+1..rgt]*, de tal forma que cada elemento del primer segmento *S[lft..pp-1]* es menor que el pivote y cualquier elemento del segundo intervalo *S[pp+1..rgt]* es mayor o igual que el pivote. Las ordenaciones recursivas de los dos subintervalos resultantes pueden ser realizadas en paralelo. La Figura 2.17 contiene el código *ll* correspondiente.

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);         (* pp contains new pivot position *)
      parallel 1..2 do
      var a, b: integer;  (* sort in parallel the two subintervals *)
      begin
        if (name = 1) then begin   a := lft; b := pp - 1;  end
        else  begin  a := pp+1; b := rgt;   end;
        sort(a, b);
      end
    end;
end; { sort }

begin
  inputArray(S); sort(1, n); writeArray(S);
end.
```
Figura 2.17 Procedimiento *sort* y programa principal

El procedimiento *part* en la Figura 2.18 realiza la división. En paralelo se compara *S[pp]* con cada elemento de *S*, y se marcan usando la variable *Mk* los elementos menores con un 1 y los mayores con un 0. Ahora necesitamos trasladar los elementos menores al principio de *S*, y los elementos mayores al final de *S*, de tal forma que el elemento *S[pp]* esté entre los dos subvectores. Esto se logra a través de una llamada al procedimiento *prefix* de la Figura 2.16 que nos devuelve en el vector *sum* las sumas parciales. La variable *NM = sum[rgt]* contiene el número de elementos menores que el pivote. Para un procesador *name* dado, la variable *sum[name]* contiene el número de elementos que están a la izquierda de *S[name]* que son menores que el pivote *S[pp]*. Esta información es suficiente para decidir en qué posición *pos* colocar cada elemento sin que existan conflictos de escritura.

2.4. Tiempo y número de procesadores

Las medidas de rendimiento de un algoritmo paralelo Λ más evidentes son el *Número de Procesadores* utilizados *P* y el *Tiempo* invertido $T_{Par}(\Lambda, P)$ por el algoritmo Λ cuando se usan *P* procesadores.

2.4.1. Suma de prefijos

Retomemos el procedimiento $\Lambda = prefix$ de la Figura 2.16 que realiza la suma de prefijos. ¿Cuántos procesadores son necesarios, supuesto que no exista virtualización de procesadores? Observe que *n* procesadores son suficientes si aplicamos el algoritmo de distribución cíclica explicado en la sección 2.2. El algoritmo distribuye la mitad de los procesadores a cada una de las dos llamadas recursivas. Estos procesadores son suficientes para la demanda que realiza la sentencia *for all* utilizada para actualizar los elementos de la parte derecha del vector *sum*.

```
shared procedure part(lft,rgt: shared integer; var pp:shared integer);
var NM, oldPp: shared integer;

shared procedure prefix(i,j: shared integer);
       .... (*ver código en la Figura 2.16 *);

begin (* part *)
  parallel lft..rgt do
    if S[name] < S[pp] then Mk[name] := 1 else Mk[name] := 0;
    prefix(lft, rgt);
    oldPp := pp;
    NM := sum[rgt];
    parallel lft..rgt do
    var pos: integer;
    begin
       if name = pp then begin
            pp := lft+NM; pos := pp
       end
       else if Mk[name] = 1 then   pos:=sum[name]+lft-1
       else begin
         pos :=   name-sum[name]+NM;
         if name < oldPp then pos:=pos+1
     end;
     S[pos] := S[name]
  end  { parallel }
end; { part }
```

Figura 2.18 Procedimiento para la partición del vector

Como vimos en la sección 2.2, el algoritmo de asignación de procesadores sólo requiere tiempo constante. Lo mismo ocurre con las operaciones implicadas de asignación a los parámetros *a* y *b* antes de la llamada, así como la actualización de la parte derecha del vector *sum*. Por tanto el tiempo empleado $T_{Par}(Prefix, n)$ es $C*log\ n$ para alguna constante *C*.

2.4.2. Ordenación Bitónica

El perfecto equilibrio de la carga de trabajo entre las tareas generadas en el algoritmo de ordenación bitónica de la Figura 2.8 hace que el algoritmo de asignación cíclica de procesadores de la sección 2.2 funcione con *n* procesadores. El procedimiento *bitonicSort* realiza *log(n)* iteraciones en las que se llama a *bSort*. Por su parte, *bSort* realiza *log(n)* llamadas paralelas recursivas. Por tanto el tiempo $T_{Par}(bitonicSort, n) = D*log^2(n)$, donde *D* es una constante.

2.4.3. Quicksort Paralelo

El tiempo invertido por el algoritmo Λ = *Sort* de la Figura 2.17 depende del número de llamadas paralelas recursivas que necesite. En cada paso recursivo se llama al procedimiento de partición. El número de etapas de partición depende del ritmo al que se reduce el tamaño de los subvectores intermedios. En un ambiente pesimista, se pueden tener particiones de tamaño muy desigual, y esto puede ocasionar que el tamaño del subvector más largo se vea decrementado en sólo una unidad después de cada iteración. Por lo tanto, en el peor caso, el algoritmo necesita *O(n)* iteraciones. Puesto que el procedimiento *part* necesita tiempo *O(log n)* para realizar la suma de prefijos, el tiempo de ejecución de *sort* puede llegar a ser tan malo como el del algoritmo secuencial *O(n log n)*. Sin embargo, si los intervalos resultantes de las llamadas a *part* son de tamaño similar, el procedimiento *sort* alcanzará sólo una profundidad logarítmica. Se puede esperar entonces que en promedio el vector esté ordenado en tiempo $O(log^2 n)$.

Puede verse fácilmente que, el algoritmo de asignación de procesadores cíclico, explicado en la sección 2.2, hace en este caso una mala gestión de los procesadores disponibles. Esto es consecuencia de que el número de procesadores requeridos en la división recursiva es proporcional al tamaño del intervalo a ordenar. Al asignar el mismo número de procesadores a los dos procesos, el algoritmo de asignación cíclica desperdicia los recursos. En el peor caso la profundidad del árbol de llamadas es *n* y se necesitarán *n(n-1)/2* procesadores.

2.5. Asignación de Procesadores Controlada por el Programador

Es claro que en el ejemplo del Quicksort paralelo una política de asignación correcta sólo requiere *n* procesadores. Existen varias alternativas para mejorar la política cíclica determinada por las ecuaciones (Ec. 2.1) y (Ec. 2.2).

```
parallel <Exp1>..<Exp2>  with weight <Exp(name)> do
  <Statement>
```

Figura 2.19 Sentencia de Asignación de Procesadores Ponderada

```
parallel <Exp1>..<Exp2>   map <Processor Mapping>  do
  <Statement>
```

Figura 2.20 Sentencia parallel con control de asignación de procesadores

La Figura 2.19 muestra una sentencia de asignación de procesadores "ponderada". En ella el programador especifica mediante pesos w_{name} la proporción de procesadores que se requerirán para el proceso *name*. La Figura 2.21 muestra su utilización en el procedimiento *sort*. En este caso el número de procesadores a asignar a cada grupo es proporcional a la longitud del intervalo.

Otra posibilidad es hacer que el programador especifique la política de asignación de procesadores. Ello significa determinar para cada procesador los elementos que definen la política (Figura 2.5). La Figura 2.20 muestra una extensión de la sentencia *parallel* del lenguaje *La Laguna* con una cláusula *map* en la que el programador en términos del valor del nombre físico del procesador (denotado por *physical*) decide a qué grupo pertenecerá cada uno de los procesadores. Utilizando esta sentencia, los códigos de la Figura 2.23 y de la Figura 2.22 computan una distribución de procesadores adaptada a la carga de trabajo. Sólo dentro del ámbito de una sentencia *<Processor Mapping>* le es permitido al programador de *ll* cambiar explícitamente las variables *numprocessors*, *name* y *physical*.

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
      a, b: shared interval;
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);         (* pp contains new pivot position *)
      a[0] := lft; b[0] := pp; a[1] := pp; b[1] := rgt;
      parallel 0..1 with weight b[name]-a[name] do
        sort(a[name], b[name]);
   end;
end; { sort }
```

Figura 2.21 Sort con asignación ponderada de procesadores

```
processor mapping proportional(left, right, pp: integer);
begin
    if (left+physical <= pp) then begin
       numprocessors := pp-left+1;
       name := 1;
    end
    else begin
      numprocessors := rgt -pp;
      name := 2;
      physical := physical-pp;
    end;
end;
```

Figura 2.22 Definiendo una política de asignación

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);      (* pp contains new pivot position *)
      parallel 1..2 map proportional(left,right,pp)   do
      var a, b: integer;
      begin
        if (name = 1) then begin   a := lft; b := pp - 1;  end
        else  begin  a := pp+1; b := rgt;   end;
        sort(a, b);
      end
    end;
end; { sort }
```

Figura 2.23 Distribuyendo los procesadores explícitamente

2.6. Las Sentencias Condicionales como Sentencias de Asignación de Procesadores

En el marco de la programación PRAM existe una extensión natural de la semántica de una sentencia condicional *if-then-else* cuando ésta tiene las dos ramas:

if <*Expression*> then < *Statement1* > else <*Statement2*>

El significado de esta extensión es que todos los procesadores en el proceso evalúan la expresión. El grupo actual de procesadores se divide en dos grupos. Los procesadores para los que la condición es verdadera forman el primer grupo y ejecutan la sentencia <*Statement1*>. Aquellos para los que la condición es falsa forman el segundo grupo y ejecutan <*Statement2*>. La sentencia condicional se convierte así en una sentencia de asignación de procesadores. Para no penalizar la eficiencia, se impone que los dos nuevos procesos se ejecutan asíncronamente (aunque dentro de cada grupo se mantiene la sincronía). Como veremos en el Capítulo 3, esta es la semántica utilizada para la sentencia condicional en *fork95*.

2.7. Aceleraciones y paradojas

Resulta interesante comparar la velocidad alcanzada por un algoritmo paralelo con la del mejor algoritmo secuencial *BestSeq* que resuelve el mismo problema. Se define la

Aceleración de un algoritmo paralelo $S(\Lambda, P)$ como el cociente entre el tiempo $T_{BestSeq}$ del mejor algoritmo secuencial y el tiempo del algoritmo paralelo $T_{Par}(\Lambda, P)$ utilizando P procesadores.

$$S(\Lambda, P) = T_{BestSeq}/T_{Par}(\Lambda, P)$$

Por ejemplo, el algoritmo para la suma de prefijos de la Figura 2.16 requiere tiempo $O(log(n))$ y utiliza n procesadores. Su aceleración es de orden

$$S(Prefix, n) = O(n/log(n))$$

El algoritmo de ordenación paralela basado en el Quicksort que aparece en la Figura 2.19 utiliza n procesadores para obtener una aceleración que, en el peor caso será de orden constante, mientras que en promedio cabe esperar una aceleración de orden $n\ log(n)/log^2(n) = n/log(n)$. Sin embargo, el algoritmo de ordenación bitónica tiene una aceleración de orden $n/log(n)$ incluso para el caso peor. No obstante, no se debe menospreciar la influencia de las constantes asociadas a los órdenes de complejidad. En general, aspiramos a desarrollar algoritmos paralelos que alcancen una aceleración tan grande como sea posible.

2.7.1. Límites en la Aceleración alcanzable

Un algoritmo paralelo es *óptimo*, cuando alcanza una aceleración de orden igual al número de procesadores que utiliza. Se dice entonces que el algoritmo alcanza una *aceleración lineal*.

Teorema 2.1 Un algoritmo paralelo no puede alcanzar una aceleración superior a la lineal:

$$S(\Lambda, P) \leq P$$

Para demostrarlo, consideremos un algoritmo paralelo Λ que se ejecuta en $T_{Par}(\Lambda, P)$ pasos usando P procesadores. El algoritmo Λ da lugar a un algoritmo secuencial Λ' que se ejecuta en una máquina secuencial en $P*T_{Par}(\Lambda, P)$ pasos. Basta que por cada instrucción de la máquina paralela la máquina secuencial ejecute un bucle de tamaño P simulando cada uno de los procesadores. Si alguno de los procesadores simulados no participa en el paso, Λ' simplemente incrementa el contador. Por definición, el mejor algoritmo secuencial *BestSeq* debe tardar menos que este algoritmo Λ' (si no fuera así, ¡*BestSeq* no sería el mejor algoritmo secuencial!):

$$T_{BestSeq} \leq T_{\Lambda'} = P*T_{Par}(\Lambda, P)$$

Despejando P se sigue que $T_{BestSeq}/T_{Par}(\Lambda, P) = S(\Lambda, P) \leq P$.

2.7.2. Aceleración Superlineal

Sin embargo, la experiencia nos muestra que los fenómenos de aceleraciones superiores a la lineal ocurren ... ¡y no tan infrecuentemente como cabría esperar después de demostrada la desigualdad anterior!. ¿Qué ocurre? ¿Es falsa la demostración anterior?

```
program SuperLinearSpeedup;
const N = 512; N1 = 511; P = 4;
type vector = array [0..N1] of integer;
var A:  shared  vector;
    x, posx, proc: shared  integer;
    found: shared boolean;
 begin
    ... (* array initialization *); x := A[(P-1)*(N div P)];
    parallel 0..P-1 do
    var first, last, k: integer;
    begin
      first := name*(N div P);
      last := first + (N div P)-1;
      k := first;  found := false;
      while (not found) and (k <= last) do
      begin
         if (A[k] = x) then begin
            found := true; posx := k; proc := name
         end;
         k := k+1;
       end;
    end;
    if found then
      writeln('found in position = ',posx,' by processor ',proc);
end.
```

Figura 2.24 ¿Puede ser la aceleración mayor que el número de procesadores?

Considere el programa *ll* de la Figura 2.24 que busca un elemento *x* en un vector desordenado. Se sabe que *x* aparece en el vector a lo sumo una vez. Bajo esta hipótesis, el algoritmo de la Figura 2.24 trabaja en una CREW PRAM. Si ocurre la especial circunstancia que el elemento *x* esta situado en la posición (*N div P)*(P-1)*, el procesador *P-1* encontrará el elemento en la primera iteración, pondrá *found* a *true* y todos los procesadores abandonarán el bucle de búsqueda. El mejor algoritmo secuencial de búsqueda (recorrer el vector de izquierda a derecha), requiere (*N div P)*(P-1)* iteraciones bajo esas hipótesis. Por ejemplo para *N = 512* y *P = 4* procesadores, el algoritmo secuencial ejecuta *384* iteraciones por una sola del paralelo. Cuando *N* es suficientemente grande la aceleración será mayor que el número de procesadores. Este fenómeno se conoce con el nombre de *Superlinealidad* o *Aceleración Superlineal*. La Figura 2.25 muestra la ejecución del correspondiente algoritmo secuencial (el programa es cargado mediante la orden *pram seq* en la línea de comandos y posteriormente ejecutado mediante la orden *go*). Son ejecutadas *17993* instrucciones del lenguaje máquina (el número que aparece tras la etiqueta *T* indica en cada instante el número de instrucciones PRAM ejecutadas). Después de reiniciar la PRAM mediante la orden *ini*, procedemos a cargar el programa ejecutable generado para el código *ll* de la Figura 2.24 (orden *load 0, speedup*). Se ejecutan *2682* instrucciones. La aceleración obtenida es por tanto igual a *17993/2682 = 6,7* que es mayor que el número *4* de procesadores utilizados. ¿Dónde está el fallo del razonamiento anterior?

Podría sospecharse que el problema es causado por la falta de realismo del modelo CREW PRAM. ¿Qué ocurre sin se trabaja con un modelo EREW?. En tal caso, la lectura concurrente a la variable *found* al comienzo del bucle de la Figura 2.24 está prohibida. Sin embargo sería posible declarar *found* como privada y sustituir la condición de finalización del bucle por el resultado de una "reducción" paralela por la operación lógica *OR* sobre las variables *found.* Tal operación de reducción puede ser realizada en tiempo logarítmico usando un algoritmo similar al explicado en la sección

2.3.2.1. El algoritmo de búsqueda resultante sigue ofreciendo una conducta superlineal en el caso particular considerado.

La verdadera respuesta reside en la imprecisión con que hemos utilizado el concepto de *Mejor Algoritmo Secuencial*. ¿En qué sentido estamos hablando del mejor algoritmo? ¿Nos referimos al caso promedio? Si es así, puede ocurrir que la versión secuencial Λ' del algoritmo paralelo sea más rápida para ciertos casos que se "desvían" del promedio y el razonamiento del Teorema 2.1 resultaría falso para esos casos. Lo mismo ocurre si estamos hablando del mejor algoritmo secuencial para "el peor caso del secuencial". Puede que dicho "peor caso" sea bastante improbable que ocurra y que la versión secuencial Λ' del algoritmo paralelo sea más veloz que el mejor algoritmo secuencial para un buen número de casos. De nuevo, para esos casos la aceleración podría ser superlineal. Esto es de hecho lo que ocurre en el ejemplo de la Figura 2.24.

Obsérvese que si un algoritmo paralelo Λ es óptimo, el orden de complejidad de su secuencialización Λ' es igual a la del mejor algoritmo secuencial *BestSeq*. Por tanto, cada vez que un diseñador de algoritmos paralelos descubre un algoritmo paralelo óptimo, nos está también ofreciendo un algoritmo secuencial que iguala en rendimiento (salvo quizá un factor constante) al mejor algoritmo secuencial conocido. Esta propiedad nos habla de la dificultad de encontrar algoritmos paralelos óptimos... y también nos sugiere que si una tabla de aceleraciones de cierto algoritmo paralelo está llena de anomalías superlineales, es bastante probable que los problemas que se utilizaron en su confección no fueran los más adecuados para el correspondiente mejor algoritmo secuencial.

El Teorema 2.1 es cierto si se fijan adecuadamente los casos del problema. Por ejemplo, si se define la aceleración como el cociente entre el tiempo del Mejor Algoritmo Secuencial sobre su Peor Caso dividido por el tiempo del Algoritmo Paralelo sobre su correspondiente Peor Caso:

$$S(\Lambda, P) = T_{BestSeq}(Worst\ BestSeq\ Case) / T_{Par}(\Lambda,\ P,\ Worst\ \Lambda\ Case)$$

o si se define la aceleración sobre el peor caso del paralelo:

$$S(\Lambda, P) = T_{BestSeq}(\ Worst\ \Lambda\ Case) / T_{Par}(\Lambda,\ P,\ Worst\ \Lambda\ Case)$$

o como el cociente de las esperanzas de las variables aleatorias $T_{BestSeq}$ y T_{Par}:

$$S(\Lambda, P) = E(T_{BestSeq}) / E(T_{Par})$$

```
C:\ >pram seq
Parallel Random Access Machine.
PRAM|T0>go
Sequential search: found in position = 384
PRAM|T17993>ini
PRAM|T0>load 0,speedup
Loaded.
PRAM|T0>go
found in position = 384 by processor 3
PRAM|T2682>_
```

Figura 2.25 Superlinealidad: Observe el contador de tiempos junto a la etiqueta *T*

2.8. El Coste, La Eficiencia y el Trabajo

El *coste* $C(\Lambda,P)$ de un algoritmo PRAM Λ se define como el producto del número de procesadores P por el tiempo empleado $T_{Par}(\Lambda, P)$

$$C(\Lambda,P) = P*T_{Par}(\Lambda, P).$$

La fórmula para $C(\Lambda,P)$ puede ser interpretada diciendo que el coste de un algoritmo paralelo es el tiempo o complejidad del correspondiente algoritmo secuencial Λ' descrito en el Teorema 2.1 El cociente $E(\Lambda,P)$ entre el tiempo del mejor algoritmo secuencial y el coste del algoritmo paralelo se denomina *eficiencia*.

$$E(\Lambda,P) = T_{BestSeq} / C(\Lambda, P)$$

La eficiencia nos da una medida del aprovechamiento de los P procesadores requeridos por el algoritmo. Un valor de $E(\Lambda,P)$ próximo a uno indica una aceleración lineal y un aprovechamiento eficiente de los procesadores.

Es obvio que el algoritmo Λ' puede ser optimizado a un algoritmo Λ'' cuyo bucle de simulación solo recorra el *número de procesadores que participan* en ese paso en vez de recorrer hasta *el número total P de procesadores necesitados*. Esto supone una mejora con respecto a Λ' porque como consecuencia de la ejecución de sentencias condicionales y bucles puede ocurrir que algunos de los procesadores no deban participar en la ejecución de ciertas instrucciones. La complejidad del algoritmo Λ'' se denomina *el trabajo*, $W(\Lambda)$, realizado por el algoritmo paralelo Λ. Obviamente el trabajo es menor que el coste: $W(\Lambda) \leq C(\Lambda, P)$.

Consideremos un programa PRAM Λ que se ejecuta en $T(P) = T_{Par}(\Lambda, P)$ pasos y requiere un trabajo $W = W(\Lambda)$. Denotaremos por $W(k)$ el número de procesadores que participan en el paso k, con $k = 0,...,T(P)-1$. Puesto que asumimos tiempo constante para las instrucciones de la máquina, el trabajo realizado en un paso k es igual al número de procesadores $W(k)$ que participan en el paso. En cada paso k, el programa Λ'' ejecuta la correspondiente instrucción $i_{k,p}$ de la máquina PRAM para cada procesador p que participa en el paso, determina si el procesador p permanecerá *participante* o no en el siguiente paso $k+1$ e incrementa o no el contador de procesadores *participantes* para el siguiente paso. Por tanto, admitiendo una complejidad constante para cada una de las instrucciones de la máquina PRAM, la complejidad W de Λ'' es:

$$W = W(0) + ... + W(T(P)-1)$$

2.8.1. Reduciendo el Número de Procesadores

Si disponemos de una máquina PRAM con P' procesadores, podemos dividir los pasos de un algoritmo paralelo Λ que usa P procesadores en dos clases de pasos, $Seq(P')$ y $Par(P')$ definidos como:

$$Seq(P') = \{k \in \{0,...,T(P)-1\} \,/\, W(k) < P'\}$$
$$Par(P') = \{k \in \{0,...,T(P)-1\} \,/\, W(k) \geq P'\}$$

El conjunto $Seq(P')$ es el conjunto de pasos en los que participan menos de P' procesadores en su ejecución. Por el contrario, el conjunto $Par(P')$ está formado por los pasos que requieren más de P' procesadores. Podemos intentar ejecutar el algoritmo PRAM Λ que utiliza P procesadores en la máquina PRAM con $P' \leq P$ procesadores, haciendo que en cada paso $k \in Par(P')$, cada uno de los P' procesadores simule aproximadamente el mismo número $W(k)/P'$ de procesadores. Así el tiempo invertido en la máquina PRAM con P' procesadores será:

$$\text{(Ec. 2.3)} \qquad T(P') = |Seq(P')| + \sum_{k \in Par(P')} ceil(W(k)/P')$$

La deducción de la fórmula (Ec. 2.3) supone cierta la siguiente hipótesis:

Hipótesis de Asignación Rápida:

Para cada paso *k* es posible calcular *W(k)* en tiempo constante y asignar los *W(k)* procesadores virtuales que resultaron participantes en el paso *k-1* a los procesadores físicos en tiempo constante.

Supuesta esta hipótesis, si se tiene en cuenta que es consecuencia directa de la definición que $|Seq(P')| \leq T(P)$, se sigue de (Ec. 2.1) la desigualdad:

$$T(P') \leq T(P)+\sum_{k\in Par(P')} ceil(W(k)/P') \leq T(P)+ceil(W/P')$$

La inecuación resultante:

$$T(P') \leq T(P)+ceil(W/P')$$

es denominada "Teorema de Brent" por algunos autores [Gib88], [Qui90] y "Work-Time Scheduling Principle" [Jaj92] por otros. Si se asume que normalmente $T(P) \leq ceil(W/P')$, se sigue que $T(P')$ es de orden $O(W/P')$. Por tanto, medido en órdenes de complejidad, la reducción de procesadores puede hacerse sin introducir ineficiencia.

2.8.2. Teorema de Brent Generalizado

La condición de que el número de procesadores *W(k)* que participan en el paso *k* pueda ser computado en tiempo constante y los *W(k)* procesadores virtuales que resultaron participantes en el paso *k-1* puedan ser asignados a los procesadores disponibles en tiempo constante no se cumple en todos los algoritmos PRAM. ¿Bajo qué condiciones es falsa esta hipótesis? Un típico ejemplo es la situación en la que un cierto conjunto síncrono de procesadores que están *participando* en un paso *k* comienzan a ejecutar el código correspondiente a una sentencia condicional sin cláusula *else*:

if <Boolean Expression> then <Statement>;

la semántica PRAM obliga a los procesadores que evalúan *<Boolean Expression>* a *FALSE* a permanecer "*ociosos*" hasta el final de la sentencia condicional. Aquí las expresiones *participante* y *ocioso* tienen significado diferentes de los de activación y desactivación utilizados en la sección 2.2. Los procesadores *ociosos* continúan "implicados" en la computación. Por ejemplo, en algunas versiones de *La Laguna*, los procesadores continúan ejecutando el código de las instrucciones sólo que los contenidos de la memoria no son modificados. Esto les sirve para seguir el "rastro" de la computación y reincorporarse sin coste alguno en el momento en el que les corresponde volver a participar. Tenemos pues tres conceptos diferentes:

1. ***Activación,***
2. ***Asignación de Procesadores y***
3. ***Participación*.**

Para ilustrar lo que ocurre cuando no se cumple la hipótesis de asignación rápida consideremos el siguiente esquema de traducción para la sentencia condicional anterior (este esquema es el utilizado por algunas versiones del compilador de *La Laguna*):

```
<Boolean Expression>.Translation
PUSHPATTERN
COPYTOPS
GOIFZERO LABEL
<Statement>.Translation
LABEL: POPPATTERN
```

Después de la traducción de la expresión lógica, *<Boolean Expression>* con la llamada a la instrucción *PUSHPATTERN*, cada procesador *p* guarda en la pila su *registro de participación*. El registro de participación es una estructura de datos en el que cada procesador "físico" almacena el "patrón" actual de los procesadores virtuales de los que se encarga que "participan" en el paso. Por ejemplo, en las mencionadas versiones de *La Laguna,* el registro de participación incluye, entre otras subestructuras, un vector de unos y ceros que indica la participación o no de los procesadores virtuales, el número de procesadores virtuales activos y un vector *NAME* conteniendo los nombres lógicos de los procesadores virtuales activos. Después de la traducción de la expresión lógica, la instrucción *COPYTOPS* "retira" los procesadores participantes que evaluaron la expresión *<Boolean Expression>* a *FALSE*. Esta instrucción *COPYTOPS* deberá contar el nuevo número $W(k+1)$ de procesadores que participan en el paso siguiente y repartir la carga de procesadores virtuales de manera equitativa entre los P' procesadores disponibles. El cálculo de $W(k+1)$ y el reparto de la carga puede hacerse mediante una suma de prefijos de las cargas individuales $W_p(k+1)$ resultantes en cada uno de los procesadores físicos $p \in \{0,...P'-1\}$. Esta suma de prefijos permite realizar posteriormente una escritura exclusiva de los nombres de aquellos procesadores virtuales que participarán en el paso $k+1$. Se trata de un algoritmo similar al utilizado en el procedimiento *part* de la Figura 2.18.

<table>
<tr><td colspan="7">———————— tiempo ————————→</td></tr>
<tr><td colspan="3">Antes de COPYTOPS</td><td></td><td></td><td colspan="2">Después</td></tr>
<tr><td>PHYS</td><td>NAME</td><td>BOOL</td><td>SUM</td><td>PREFIX</td><td>PHYS</td><td>NAME</td></tr>
<tr><td rowspan="4">0</td><td>2</td><td>1</td><td rowspan="4">2</td><td rowspan="4">2</td><td rowspan="3">0</td><td>2</td></tr>
<tr><td>4</td><td>0</td><td>7</td></tr>
<tr><td>7</td><td>1</td><td>9</td></tr>
<tr><td>8</td><td>0</td><td rowspan="3">1</td><td>10</td></tr>
<tr><td rowspan="4">1</td><td>9</td><td>1</td><td rowspan="4">2</td><td rowspan="4">4</td><td>15</td></tr>
<tr><td>10</td><td>1</td><td>19</td></tr>
<tr><td>11</td><td>0</td><td rowspan="3">2</td><td>21</td></tr>
<tr><td>13</td><td>0</td><td>33</td></tr>
<tr><td rowspan="4">2</td><td>15</td><td>1</td><td rowspan="4">4</td><td rowspan="4">8</td><td>35</td></tr>
<tr><td>19</td><td>1</td><td rowspan="3">3</td><td>46</td></tr>
<tr><td>21</td><td>1</td><td>51</td></tr>
<tr><td>33</td><td>1</td><td>62</td></tr>
<tr><td rowspan="4">3</td><td>35</td><td>1</td><td rowspan="4">4</td><td rowspan="4">12</td><td></td><td></td></tr>
<tr><td>47</td><td>1</td><td></td><td></td></tr>
<tr><td>51</td><td>1</td><td></td><td></td></tr>
<tr><td>62</td><td>1</td><td></td><td></td></tr>
</table>

Figura 2.26 Ejecución de COPYTOPS en una PRAM con 4 procesadores

La Figura 2.26 muestra una traza de COPYTOPS en una PRAM con $P' = 4$ procesadores que virtualizan $P = 16$ procesadores lógicos. La columna denominada BOOL contiene el resultado de la evaluación de *<Boolean Expression>*. Las entradas de la columna SUM muestran los números de procesadores lógicos por procesador físico (los nombres de estos últimos aparecen en las columnas etiquetadas PHYS) que participarán en el siguiente paso. Por ejemplo, el procesador físico *1* está al cargo de la simulación de un cierto número de procesadores lógicos, de los cuáles están participando los procesadores lógicos *9, 10, 11* y *13*. Los procesadores lógicos *9* y *10* evalúan la expresión a cierta y los otros dos a falsa. Después de la suma de prefijos (columna PREFIX) de los elementos del vector SUM, cada procesador físico compacta en el vector NAME los nombres de sus procesadores lógicos participantes. Lo hace, como indican las flechas en la Figura 2.26 en las posiciones a partir de PREFIX[PHYS-1]. Mediante un cálculo sencillo los procesadores físicos pueden determinar cuántos procesadores lógicos corresponden a cada uno y de qué segmento del vector NAME se deben de encargar. **Así pues hay un coste adicional *O(log(P'))* en aquellos pasos que implican un descenso en el número de procesadores que participan. Este tiempo *O(log(P'))* es necesario para poder calcular *W(k+1)* y asignar las tareas para el paso *k+1* a los procesadores.** La ecuación (Ec. 2.4) se escribe bajo esta consideración como:

$$T(P') \leq |Seq(P')| + \sum_{k\in Par(P')} log(P') + ceil(W(k)/P')$$

de donde se deduce

$$T(P') \leq |Seq(P')| + log(P') * |Par(P')| + ceil\,(W/P')$$

El teorema de Brent sigue siendo válido si se expresa en términos de órdenes de complejidad, esto es *T(P')* es de orden

$$O(T(P)+ceil(W/P'))$$

Como resultado de la acción de *COPYTOPS* puede ocurrir que la evaluación de la expresión lógica resulte falsa para todos los procesadores y que ningún procesador resulte participante. En tal caso la instrucción *GOIFZERO LABEL* hace saltar a todos los procesadores a la instrucción *POPPATTERN* en la que se recupera el antiguo patrón de procesadores participantes. En caso contrario, se ejecuta el código resultante de la traducción de la sentencia.

2.8.3. La Ley de Amdahl Generalizada

De la definición de los conjuntos *Seq(P')* y *Par(P')* se sigue que

$$T_{Par}(\Lambda, P') \leq |Seq(P')| + |Par(P')|\ /P'$$

Sustituyendo en la definición de la aceleración tenemos:

$$S(\Lambda, P') = T_{BestSeq}/T_{Par}(\Lambda, P') \leq T_{BestSeq} /\{|Seq(P')| + |Par(P')|\ /P'\}$$

Como $|Par(P')| = W - |Seq(P')| \geq T_{BestSeq} - |Seq(P')|$, resulta que:

$$S(\Lambda, P') \leq 1 /\{|Seq(P')| / T_{BestSeq} + (T_{BestSeq} - |Seq(P')|)/(T_{BestSeq} * P')\}$$

Si denotamos por f el porcentaje $f = |Seq(P')| / T_{BestSeq}$ de operaciones que requieren menos de P' procesadores con respecto al número de operaciones que realiza el mejor algoritmo secuencial, obtenemos la inecuación:

$$S(\Lambda, P') \leq 1/(f+(1-f)/P')$$

La desigualdad anterior se conoce con el nombre de ley de Amdahl. Obsérvese que como corolario se sigue que $S(\Lambda, P') \leq 1/f$. Por tanto, la aceleración alcanzable por un algoritmo Λ sobre una máquina con P' procesadores está acotada por el porcentaje $W / |Seq(P')|$ de operaciones (con respecto al trabajo) de pasos que requieren menos de P' procesadores. En particular $W / |Seq(1)|$ es una cota inferior de la aceleración alcanzable por Λ sobre cualquier máquina paralela. Por ejemplo, si el *25%* de las operaciones de un algoritmo Λ están en *Seq(1)*, la aceleración nunca podrá ser mayor que *4*, ya que

$$4 = 100/25 = W/|Seq(1)| \geq T_{BestSeq} / |Seq(1)|$$

2.9. Simulación del Modelo PRAM por una Mariposa

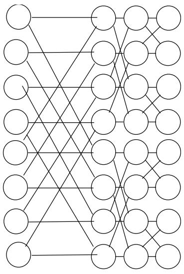

Figura 2.27

En este apartado discutiremos algunos de los problemas que surgen en el diseño de los accesos a variables compartidas en la realización de un lenguaje orientado al modelo PRAM en un sistema distribuido. Estos problemas fueron tratados en [Leo97] y [Leo95a]. La discusión se realiza sobre una mariposa, pero los razonamientos son válidos para la mayoría de las redes "hipercúbicas".

Una mariposa o *"butterfly"* N-dimensional tiene $(N+1)*2^N$ nodos y $N*2^{N+1}$ aristas. Los nodos se corresponden con pares (w,i) donde i es el *nivel* o *dimensión* del nodo y w es un número que denota la *fila* del nodo.

Dos nodos (w,i) y (w',i') están conectados si, y sólo si

$$(i' = i+1 \wedge w = w') \vee (i' = i+1 \wedge \mathrm{w} = \mathrm{w'}\ \mathrm{XOR}\ 2^{\mathrm{i}})$$

Es costumbre convertir la mariposa en una topología cilíndrica haciendo que las columnas inicial y final coincidan.

Existe una correspondencia natural entre cada nodo del hipercubo y cada fila de la mariposa. La Figura 2.27 muestra una mariposa con 3 dimensiones y 8 filas. Se puede obtener un hipercubo de una mariposa simplemente colapsando los nodos en la misma fila y observando cómo se solapan los enlaces proyectados. Recíprocamente, se puede obtener una mariposa de un hipercubo desplegando cada nodo en una fila de $N+1$ nodos y haciendo que cada uno de estos $N+1$ procesadores se ocupe de un enlace en una dimensión diferente. Cada uno de los nodos desplegado de una fila se enlaza con el nodo siguiente en la fila correspondiente a la dimensión de la que se hace cargo. De aquí se sigue que un hipercubo con N nodos puede ser simulado en $log(N)$ pasos por una mariposa con $N*(log(N)+1)$ nodos. La mariposa tiene un grado de 4, un diámetro $O(log(N))$ y un ancho de bisección $\Theta(N/log(N))$.

Teorema 2.2 /Construcción

Cualquier algoritmo Λ que se ejecuta en tiempo $T = T_{Par}(\Lambda,P)$ en una *(P,M)-EREW PRAM* con P procesadores y M celdas de memoria compartida puede traducirse a un algoritmo Λ^b que se ejecuta en tiempo $O(T*(P/P'+log(P')))$ en una máquina mariposa (butterfly) con $P' \leq P$ procesadores con una probabilidad próxima a uno.

Esquema de la demostración/implementación: Supongamos que $P' = 2^r$ y que $M = 2^s$. Cada uno de los P' procesadores en la mariposa emula a un conjunto P/P' de los P procesadores de la *(P,M)-PRAM*. Cada procesador se encarga también de la emulación de M/P' celdas de memoria compartida usando para ello parte de su memoria privada. El problema principal es ¿Qué hacer si algunos de los P procesadores PRAM acceden a la memoria compartida?. La solución pasa por disponer de una función h^Λ

$$h^\Lambda: [0,M-1] \longrightarrow [0,P'-1]\times[0,M/P'-1]$$
$$h^\Lambda(m) = (h^\Lambda_P(m), h^\Lambda_M(m)) \in [0,2^r-1]\times[0,2^{s-r}-1]$$

Esta aplicación puede ser generada por el compilador a partir del programa PRAM Λ El valor $h^\Lambda(m)$ para una dirección m de memoria compartida dada, nos determina el procesador $h^\Lambda_P(x)$ que la guarda y la dirección $h^\Lambda_M(x)$ en la que está almacenada en dicho procesador $h^\Lambda_P(x)$. Entonces cada procesador físico, según avanza en el bucle de tamaño P/P' de virtualización va lanzando las demandas del correspondiente procesador virtual utilizando la función h^Λ. De esta manera se solapan cómputo y comunicaciones. El paralelismo de segmentación que se produce hace que el tiempo de latencia $O(log(P'))$ marcado por el diámetro de la mariposa sólo tenga que ser contabilizado una vez. La función h^{Λ} puede construirse a partir una función g_{Λ} biyectiva en el espacio de las direcciones de memoria compartida:

$$g_{\Lambda:}\ [0, M\text{-}1] \longrightarrow [0, M\text{-}1]$$

y definiendo las funciones componentes h_P y h_M de h en el espacio de nombres de procesador y de direcciones de memoria como los números que cumplen la ecuación:

$$g\Lambda\ (m) = hM(m)*2r + hP(m)$$

o lo que es lo mismo:

$$hM(m) = g\Lambda\ (m)\ div\ 2r \quad y\ hP(m) = g\Lambda\ (m)\ mod\ 2r$$

La función g_{Λ} es construida por el compilador adaptada a los patrones de acceso del algoritmo Λ de manera que los accesos resulten lo más uniformemente distribuidos y la congestión sea escasa. Una mejora adicional es hacer que el compilador calcule una función $g_{\Lambda,A}$ distinta adaptada a los patrones de acceso a la variable A declarada como compartida en el programa PRAM Λ. Lo ideal es que el cómputo de $g_{\Lambda,A}$ pueda hacerse en tiempo constante. Normalmente, la construcción de la función $g_{\Lambda,A}$ está basada en técnicas de "hashing". En ocasiones una combinación de "mappings" cíclicos y por bloques como las utilizadas en algunos lenguajes paralelos es suficiente [Hpf93].

El problema de encontrar la asignación de una variable a los procesadores de manera que se minimice la congestión en los procesadores puede formalizarse como sigue. Para un paso PRAM s y un procesador i denotamos por $h_{s,i}$ el valor:

$$\text{(Ec. 2.5)} \qquad h_{s,i}\ (g_{\Lambda,A}) = max\ \{in_{s,i}\ (g_{\Lambda,A}),\ out_{s,i}\ (g_{\Lambda,A})\}$$

donde

- $in_{s,i}(g_{\Lambda,A})$ es el número de peticiones a la memoria compartida administrada por el procesador i de la mariposa en el paso s para la distribución de memoria fijada $g_{\Lambda,A}$
- $out_{s,i}(g_{\Lambda,A})$ es el número de peticiones a la memoria compartida realizadas por el procesador i de la mariposa en el paso s, para la distribución de memoria $g_{\Lambda,A}$

El operador de máximo en (Ec. 2.5) puede ser sustituido por el de suma según sea el número de puertos y la capacidad de entrada/salida paralela que tenga la red.

El problema es buscar la función g dentro de una familia Ψ de funciones analíticas cuyo cómputo pueda ser implementado de manera muy eficiente (en tiempo constante) y tal que minimice el desequilibrio de la congestión en los procesadores de la mariposa:

$$\text{(Ec. 2.6)} \qquad min_{g\in\Psi} \sum_{s=0,..,R} max_{\ i=0,..,P-1} |h_{s,i}(g) - E(h_s(g))|$$

donde R denota el número de pasos del programa PRAM y $E(h_s(g))$ es el promedio de peticiones a memoria en el paso s.

$$E(h_s(g)) = (\textstyle\sum_{i=0,..,P-1} h_{s,i}(g))/P$$

Si la variable A implicada en los accesos a memoria en el paso es relativamente pequeña, puede ser factible renunciar a que g se pueda expresar en forma analítica y duplicar la cantidad de memoria requerida para A, utilizando para su implementación una estructura de punteros a procesadores.

La resolución del problema (Ec. 2.6) resuelve también parte del problema de equilibrio de la carga de trabajo entre los procesadores planteado en la sección 2.5.

Obsérvese que en esta simulación la asignación de los procesadores PRAM a los procesadores de la mariposa es estática y se determina al comienzo de la computación. Puede ocurrir que los procesadores asignados a algunos procesadores físicos de la mariposa dejen de participar temporalmente en la ejecución como consecuencia de la evaluación de condiciones lógicas en sentencias condicionales y bucles, mientras los procesadores PRAM de otros procesadores físicos de la mariposa se mantienen participando. De esta manera se produciría un desequilibrio en la carga de trabajo. El problema de encontrar una buena asignación de los procesadores PRAM a los procesadores físicos de la mariposa está asociado al problema de la asignación de las direcciones de memoria compartida a los procesadores.

Se puede demostrar que para cada programa PRAM Λ y cada variable EREW A existen funciones $g_{\Lambda,A}$ que garantizan que el envío de las peticiones y las respuestas está libre de congestiones con una probabilidad próxima a uno [Lei92]. Por tanto los accesos pueden realizarse en tiempo $O(P/P' + log(P'))$. Como el algoritmo PRAM simulado requiere T pasos, tenemos una complejidad $O(T*(P/P'+log(p')))$.