fork95 y la SB-PRAM

Por Francisco de Sande González

3.1. Introducción

Cuando a principios de la década de los 90 comenzamos a trabajar en el diseño del lenguaje *ll*, éste era la única implementación de un lenguaje con todas las características exigibles a un lenguaje orientado al modelo PRAM [Leo92]. En 1995 surge *fork95*. Junto con *ll*, este lenguaje soporta recursividad y paralelismo. Las soluciones ofrecidas en su diseño e implementación constituyen alternativas para el caso de memoria compartida, a las propuestas que hacemos en este trabajo. El proyecto en el que está inscrito *fork95* implica también el desarrollo de un prototipo de arquitectura PRAM, denominado SB-PRAM.

En la sección 3.2 de este Capítulo presentamos las principales características de la SB-PRAM y el epígrafe 3.3 está dedicado al estudio de las características de *fork95* como lenguaje orientado al modelo PRAM.

3.2. La SB-PRAM

La SB-PRAM [Sbp], [Abo93], es un multiprocesador MIMD escalable, masivamente paralelo fuertemente síncrono, de memoria compartida con tiempo de acceso a memoria uniforme. El proyecto está siendo desarrollado en el Departamento de Informática de la Universidad de Saarbrücken (Alemania) por el equipo del profesor W. J. Paul y lleva en marcha desde principio de los años 90. El objetivo del proyecto es conseguir una máquina de 64 procesadores físicos (2048 virtuales) de tipo RISC y una memoria compartida de 2Gb.

El proyecto SB-PRAM ha sido concebido como un proyecto universitario de bajo presupuesto que pretende probar que es posible la implementación de una PRAM, es por ello que en este estudio haremos hincapié en explicar las ideas en las que se basa la implementación más que en una presentación de las prestaciones que es posible lograr. En la actualidad está disponible un prototipo, la 4-SB-PRAM con cuatro procesadores físicos y una velocidad de procesador de 0.25 MFlops. Los autores del proyecto indican en [Bac97] las modificaciones necesarias para pasar a otro prototipo esta vez con 128 procesadores físicos que se encuentra actualmente en construcción y en [For96] presentan HPP, una tercera implementación utilizando la última tecnología VLSI y enlaces de alta velocidad así como otras alternativas tecnológicas. La nueva máquina, HPP correría a una velocidad de 96MHz y alcanzaría un incremento de un factor 10 con respecto a una SB-PRAM de 128 procesadores. Con las prestaciones de los desarrollos futuros de la SB-PRAM los autores han comprobado mediante simulaciones que es posible obtener para algunos benchmarks significativos unas prestaciones sostenidas similares a las de multiprocesadores de memoria distribuida comerciales actuales.

La SB-PRAM está formada por *n* procesadores y un número igual de módulos de memoria. Como en todas las máquinas de memoria compartida, los procesadores están interconectados con los módulos de memoria por una red de interconexión (ver Figura 3.1) que en el caso de la SB-PRAM tiene topología de mariposa. Un acceso a la memoria en un procesador se transforma en un paquete de datos que es inyectado en la red de interconexión y viaja hasta el módulo de memoria adecuado. En caso que se trate de una operación de lectura de la memoria, un paquete de datos de respuesta viaja en

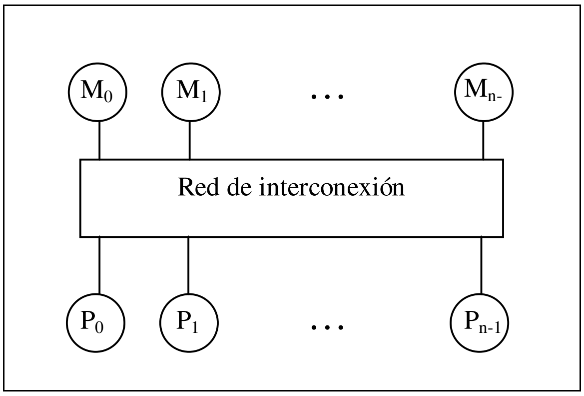

Figura 3.1 Implementación de una PRAM

sentido inverso desde el módulo de memoria hasta el procesador.

Para conseguir altas prestaciones en una arquitectura de este tipo se debe poner especial cuidado en el diseño de ciertos elementos. En particular, disminuir (u ocultar) la latencia de la red de interconexión es un objetivo crucial del diseño. La latencia se oculta mediante dos estrategias: en primer lugar mediante la virtualización de procesadores, por la cual cada procesador físico de la máquina simula el comportamiento de un cierto número de procesadores virtuales (procesos o hebras de ejecución). En segundo lugar, se utiliza una instrucción LOAD retardada: la red no proporciona inmediatamente la respuesta a una instrucción LOAD. Otro aspecto importante es limitar la congestión de los módulos de memoria (puntos calientes). Si en un determinado instante del cómputo todas las referencias a memoria están soportadas en el mismo módulo, los accesos a este módulo de memoria se convertirán en un cuello de botella. La congestión de los módulos se evita también usando dos estrategias: en primer lugar, las direcciones de memoria producidas en los procesadores se transforman mediante una función hash que las distribuye uniformemente entre los diferentes módulos de memoria, y por otra parte, los nodos de la red combinan paquetes de datos con la misma dirección de destino en un único paquete, disminuyendo el tráfico de información en la red. Con esta estrategia de la combinación también se implementa en hardware las operaciones de prefijos.

Las ideas subyacentes al desarrollo de la SB-PRAM no han sido desarrolladas por primera vez en este proyecto. El concepto de procesadores virtuales en hardware se utilizó en el Denelcor HEP [Smi78] o en el Tera MTA [Alv90]. Las ideas del hashing y de la combinación de paquetes se habían utilizado ya en el NYU Ultracomputer [Got83] y la IBM RP3 [Pfi85]. El NYU Ultracomputer, el IBM RP3, la Tera MTA y el Stanford DASH [Len92] (un multiprocesador con memoria virtual compartida y coherencia de cache) también tienen soporte hardware para operaciones de prefijos. El propósito de la SB-PRAM consiste en implementar todas estas ideas en una única máquina.

La SB-PRAM consta de *n* procesadores físicos que implementan una priority CRCW-PRAM con $p=v*n$ procesadores, siendo *v* el número de procesadores virtuales por procesador físico. La arquitectura está basada en la de la Fluent Machine de Ranade [Ran88]. La Figura 3.2 presenta una visión global de la arquitectura de la SB-PRAM. El diseño de la máquina ha precisado del desarrollo de tres circuitos integrados específicos: el procesador, los nodos de la red de interconexión y el chip de ordenación que en la Figura 3.2 aparece con la etiqueta 'sorter'. En nuestra descripción prestaremos atención a aquellas características de la arquitectura que posibilitan la implementación eficiente de una PRAM, y no abordaremos otras decisiones de diseño que son ampliamente utilizadas en otro tipo de máquinas.

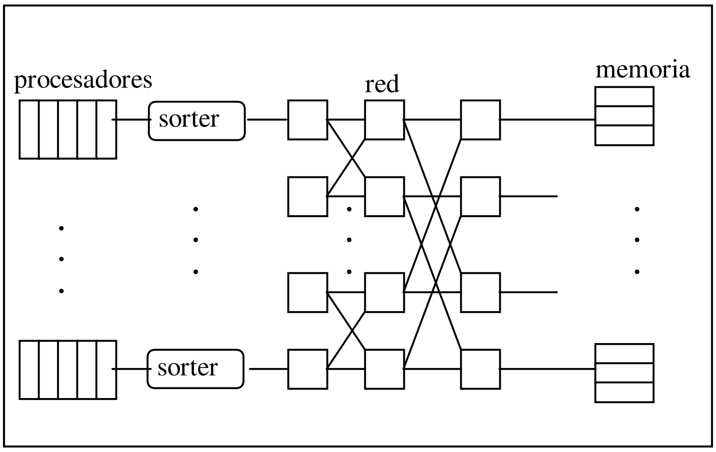

Figura 3.2 Arquitectura de la SB-PRAM

3.2.1. Los procesadores

El procesador [Kel94] implementa una arquitectura de tipo multihilo (multithreaded) Berkeley-RISC-1 con formato de instrucciones de 32 bits. Soporta en hardware operaciones aritméticas y lógicas sobre operandos enteros de 32 bits, así como suma y producto (no división) en punto flotante en simple precisión de acuerdo al estándar IEEE. Las instrucciones son de tres direcciones y las aritméticas pueden operar solamente sobre registros. La SB-PRAM suministra también como instrucciones máquina operaciones de prefijo para suma, máximo, Y lógico, y O lógica que se ejecutan en 2 ciclos de CPU.

Cada procesador físico soporta 32 procesadores virtuales. Cada procesador posee su propia memoria de programa (arquitectura Hardvard), con lo que se reduce el tráfico que introduciría en la red el almacenar el programa en la memoria compartida. Con una memoria de programa por cada procesador físico, todos los procesadores virtuales ejecutan el mismo programa; no obstante, es posible hacer bifurcaciones dependientes del número del procesador virtual.

Un procesador virtual está caracterizado por el contenido de sus registros específicos mientras que la unidad aritmético lógica y la unidad de control y decodificación son compartidas por todos los procesadores virtuales de un procesador físico. Los procesadores virtuales se planifican de modo cíclico. Después de la ejecución de una instrucción, el control se transfiere al siguiente procesador virtual (proceso), independientemente de la ocurrencia de accesos a memoria. Este tipo de control requiere un cambio de contexto después de la ejecución de cada instrucción (para el cual hay hardware específico) y exige que todas las instrucciones conlleven el mismo tiempo de ejecución. Llamaremos un *ciclo de procesador* a una vuelta en el bucle de planificación de los 32 procesadores virtuales de un procesador físico. Cada procesador virtual posee su propio conjunto de registros: 32 registros de propósito general que se ubican fuera del chip y 6 de propósito específico. Además, cada procesador físico (y por tanto todos sus procesadores virtuales) tienen acceso a cuatro registros de propósito especial que se utilizan en la transformación de las direcciones generadas por el procesador. El procesador utiliza operaciones LOAD y STORE diferentes para acceder a su memoria local o global. Los accesos locales se utilizan para accesos a memoria y periféricos ubicados en la misma placa que el procesador, mientras que los globales son enviados a la red de interconexión. La instrucción LOAD global se implementa con un retardo unitario, es decir, el registro destino de una instrucción LOAD se actualiza después que ha sido ejecutada la instrucción siguiente al LOAD. Los autores del diseño afirman que en la mayoría de los casos no es necesaria la inserción de instrucciones de no-operación después de las LOAD. No existen caches en el chip y el procesador opera a una frecuencia de reloj de 8MHz.

Cuando se realiza un acceso a la memoria compartida, las direcciones lógicas se transforman en direcciones físicas en dos pasos. En primer lugar, si el bit más significativo de la dirección lógica es uno, se suma a ésta el contenido de un registro especial *hi_base*; en caso contrario se suma el contenido del registro *lo_base*. La finalidad de esta operación es asignar diferentes espacios de memoria a datos privados y compartidos. El segundo paso consiste en aplicar una función hash de la forma $H(x) = a\, x \bmod m$ donde x es la dirección transformada que se obtuvo en el paso anterior, y m, el tamaño de la memoria se supone que es una potencia de dos. El factor a es un entero impar en el rango [1, m-1] que se elige aleatoriamente antes del comienzo de la ejecución de la aplicación y se almacena en un registro especial dentro del chip. El módulo de memoria $h(x)$ que contiene la dirección x viene determinado por

$$h(x) = \lfloor H(x)/(m/p) \rfloor$$

siendo p el número de módulos de memoria (que coincide con el número de

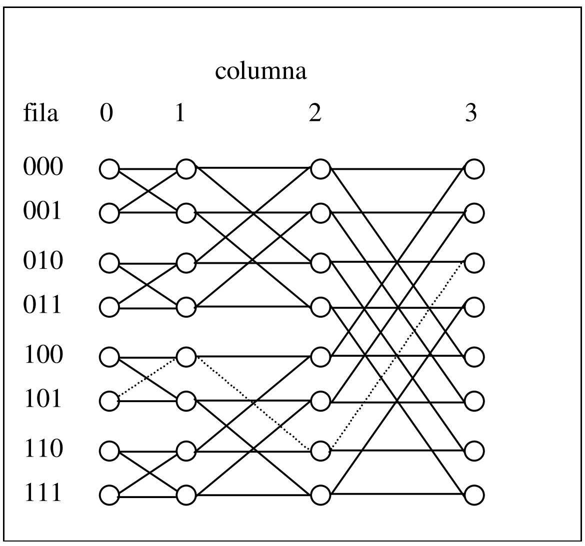

Figura 3.3 Red Mariposa de 4 etapas

procesadores físicos). Por lo tanto, el número del módulo de memoria se almacena en los *log p* bits más significativos de la representación binaria de *H(x)*. Los bits menos significativos indican la dirección dentro del módulo. La función hash utilizada tiene las ventajas de ser muy fácil de computar (el módulo se limita a la extracción de un cierto número de bits de un registro, y el producto se realiza en registros específicos) y de ser una función biyectiva, lo cual facilita la implementación del algoritmo de encaminamiento de información en la red de interconexión. Los autores aseguran que con esta función hash simple consiguen unos resultados suficientemente buenos [Eng93].

Los procesadores no utilizan ningún tipo de memoria cache (que es la aproximación que utiliza el DASH o la KSR [Alm94] debido a que ello provoca variaciones en el tiempo de acceso a memoria, con la dificultad añadida del mantenimiento de la coherencia de las caches.

La finalidad del chip de ordenación (sorter) consiste en ordenar por sus direcciones las peticiones de acceso a memoria que realiza cada uno de los procesadores, de forma que estas peticiones se inyectan en la red de interconexión en orden ascendente de direcciones. El sorter ejerce de interface entre el procesador y la red y hay uno por cada procesador físico. El chip de ordenación implementa un array lineal sistólico de 8 entradas que puede ser llenado y vaciado simultáneamente. El chip se encarga de mezclar en uno sólo todos los accesos a una misma posición de memoria que se produzcan dentro del mismo *ciclo de procesador*. El chip trabaja a una frecuencia de reloj de 32MHz, que es también la frecuencia de trabajo de los nodos de la red de interconexión.

3.2.2. La red de interconexión

La red de interconexión de la SB-PRAM tiene una topología de mariposa. Como ya estudiamos en el epígrafe 2.9 del Capítulo 2, una red mariposa de grado 2 consiste en una red de $n(1+\log n)$ nodos. Cada nodo tiene asignado un número unívoco $<col, fil>$ con $0 \le col \le \log n$, $0 \le fil \le n-1$. $<col, fil>$ puede verse como la concatenación de las representaciones binarias de los números *col* y *fil*. El nodo $<col, fil>$ está conectado con los nodos $<col+1, fil>$ y $<col+1, fil \otimes 2^{col}>$ donde $\otimes$ denota la o-exlusiva. La Figura 3.3 presenta una red mariposa de 4 etapas. La Figura 3.4 representa la red de mariposa de la SB-PRAM, con cuatro entradas y cuatro salidas.

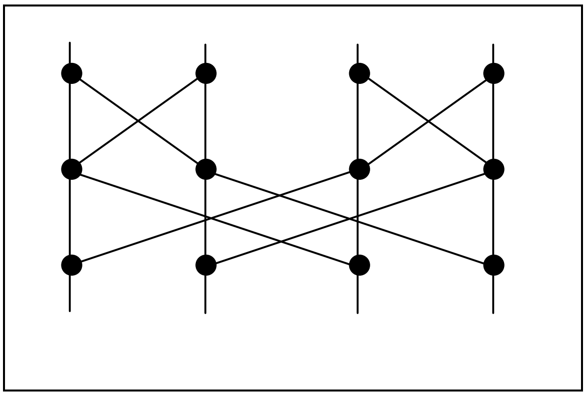

Figura 3.4 Topología de la red de la 4-SB-PRAM

Los nodos de la red de interconexión [Cro93] cuyos chips han sido específicamente diseñados para la máquina implementan un nodo de encaminamiento de una mariposa con 2 entradas, in_0 e in_1 y 2 salidas out_0 y out_1 (en la Figura 3.6 se presentan dos de estos nodos).

Supondremos que in_0 y out_0 son los enlaces que para un determinado nodo aparecen en la dimensión horizontal mientras que in_1 y out_1 son los de la dimensión no-horizontal. La forma de encaminar datos en una red mariposa es simple: en un determinado nodo se compara el bit de la columna de ese nodo con el mismo bit del nodo destino; si son iguales, el mensaje ha de encaminarse por out_0, en caso contrario por out_1. En la Figura 3.3 se representa con trazo discontinuo el camino que seguiría un mensaje que partiera del nodo <5, 0> con destino al nodo <2, 3>.
Los procesadores envían peticiones de lectura, escritura o prefijos a la memoria, y estas peticiones son encaminadas hacia los módulos de memoria a través de la red de interconexión. Si la petición es una lectura (LOAD) o una operación de prefijo será necesario enviar una respuesta a través de la red al procesador que realizó la petición. Los paquetes de datos que circulan por la red consisten en un modo (READ, WRITE o PREFIX_OP), una dirección y en el caso de WRITE o PREFIX_OP un dato. Para obtener una longitud de paquete única se inserta un valor que no será utilizado (dummy) en los paquetes READ.

Cada procesador está conectado a un nodo de entrada de la red a través de un chip de ordenación. El sorter recolecta las peticiones de memoria de 8 procesadores virtuales y las inyecta en la red ordenadas por su dirección de destino. Después de enviar las peticiones el sorter inyecta en la red un paquete especial con modo EOR (End of Round) y dirección de valor infinito, que indica el final de lo que llamaremos un *ciclo de red*. El paquete EOR se inyecta también en la red en el caso que no haya peticiones de acceso a memoria en un determinado *ciclo de procesador*. Dado que cada procesador físico emula 32 procesadores virtuales, en un *ciclo de procesador* tienen lugar cuatro *ciclos de red* y se inyectarán en la red 4 paquetes EOR por procesador. Si se requiere un paquete de datos respuesta, se almacena información de encaminamiento en una cola FIFO en cada nodo de la red. En el sentido desde los procesadores hacia la memoria, el encaminamiento se determina por la dirección de la petición, mientras que en el sentido de la memoria hacia los procesadores, el camino se determina mediante la información que se almacenó en las colas en el camino 'de ida'.

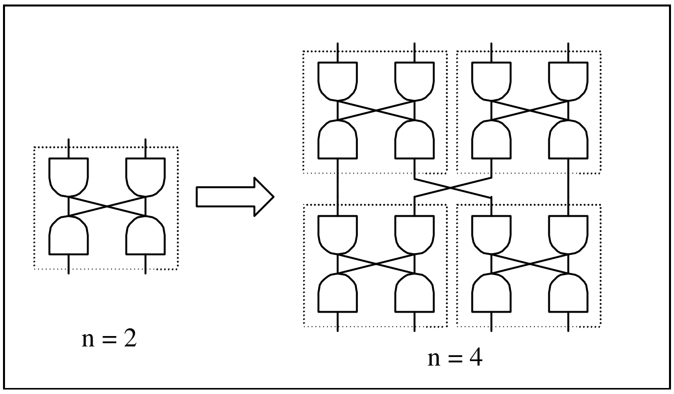

Figura 3.5 Particionado de la red

La red ha de tener una frecuencia de reloj al menos doble de la de los procesadores debido a que las partes de direcciones y datos de un paquete de datos están multiplexadas. Por simplicidad se eligió una frecuencia de reloj cuatro veces más alta que la de los procesadores. Los nodos de la red se han diseñado en la misma tecnología que el procesador (tecnología *sea of gates*) y tal como se muestra en la Figura 3.5, partes de diferentes nodos se han integrado en un único chip. En la parte izquierda de la figura se presenta un esquema del contenido de un chip, mientras que en la parte derecha se muestra cómo conectando entre sí los chips se obtiene la red de la 4-SB-PRAM (Figura 3.4). Esta decisión del diseño ha venido motivada por el esfuerzo en reducir el coste de cableado fuera del chip así como la reducción del número de circuitos a utilizar [Cro93]. Los autores del diseño indican [Bac97] que una primera versión del chip de los nodos de la red que se llegó a construir se descubrió que era incorrecto debido a una incorrecta implementación del algoritmo de encaminamiento. Debido a ello, en el prototipo disponible de la 4-SB-PRAM la red es emulada con un diseño FPGA (utilizan la FPGA XC4013 de Xilinx) con una frecuencia de reloj de 5MHz. Ello les ha permitido comprobar la corrección del diseño, y cuando construyan el chip, éste tendrá la frecuencia prevista de 32MHz.

3.2.2.1. El algoritmo de ruteo

El algoritmo de encaminamiento que utiliza la SB-PRAM [Kel95] es una versión segmentada del algoritmo de Ranade [Ran91] con colas FIFO de tamaño 16. La idea fundamental de este algoritmo (que es la misma que la que utiliza la Fluent Machine de Ranade) consiste en conseguir que la red preserve el orden de las peticiones de acceso a memoria que produjo el chip de ordenación a la salida del procesador. Cada nodo de la red selecciona de sus dos buffers de entrada (ver Figura 3.6) el paquete con menor dirección para ser emitido a la salida, manteniendo el orden de los paquetes de entrada (dos secuencias ordenadas que se mezclan produciendo una única secuencia, también ordenada). Si en un mismo *ciclo de red* hay peticiones de memoria del mismo tipo (lectura, escritura, prefix_op) para la misma dirección de memoria, las peticiones se combinan y una sola petición se pasa a la siguiente etapa de la red. Esto es posible porque la red, como hemos indicado, preserva la ordenación establecida por el chip de ordenación. En el camino de vuelta (sentido desde memoria hacia el procesador) las peticiones que fueron combinadas se re-duplican utilizando la información que se almacenó en las colas. Debido a la combinación de peticiones a memoria, a lo sumo 4 peticiones a una misma dirección de memoria alcanzan un determinado módulo de memoria incluso si todos los procesadores virtuales de la máquina acceden simultáneamente la misma posición de memoria. Dado que la SB-PRAM implementa

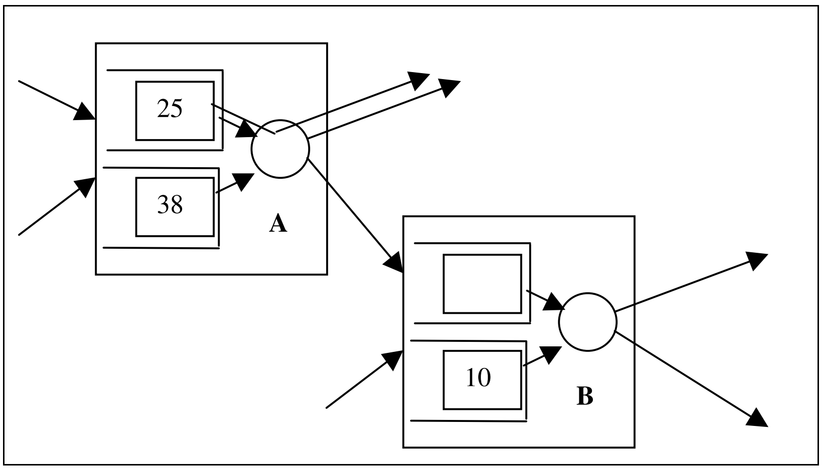

Figura 3.6 Utilización de los paquetes fantasma en los nodos de la red

una priority-CRCW PRAM, una petición de escritura concurrente se resuelve seleccionando el operando correspondiente al procesador con mayor número, que se puede determinar a partir del enlace por el que llega la petición al nodo. Las operaciones de prefijo (multiprefijo) se implementan en la red de interconexión como una forma especial de combinación. En el camino desde procesadores a memorias dos operandos *i* y *j* se procesan de acuerdo a su operación de prefijo *op* ∈ *{and, or, max, add}* y el resultado se envía en el sentido hacia la memoria. Además, el operando, *op* y el operador izquierdo *i* se almacenan en otra cola FIFO. En el camino desde la memoria hacia el procesador, el resultado entrante en un nodo, *s* y *s op i* son enviados hacia los procesadores (uno por cada enlace en el sentido memoria hacia procesadores). Los nodos de la red de interconexión disponen de una unidad aritmético lógica para llevar a cabo estas operaciones.

El paquete de datos seleccionado en un nodo se transmite al siguiente nivel de la red a través del enlace de salida adecuado. Sólo los paquetes con modo EOR se emiten por las dos salidas del nodo para garantizar la separación de los *ciclos de red*. La dirección infinita de los paquetes EOR garantiza que un paquete EOR sólo será seleccionado si ambos buffers de entrada contienen paquetes EOR.

Consideremos la situación representada en la Figura 3.6 en la que el nodo B no puede transmitir el paquete con dirección 10 porque debe asegurar que no recibirá dentro del mismo *ciclo de red* un paquete con dirección menor que 10. Cuando el nodo A selecciona el paquete con destino 25 para transmitirlo a través del enlace de salida superior, puede transmitir esa información al nodo B enviándole un paquete con un modo especial GHOST (fantasma) y dirección 25 por el otro enlace de salida. Cuando el nodo B recibe el paquete fantasma sabe que debido al orden establecido, todos los paquetes que recibirá en el futuro por el enlace por el que ha recibido el paquete fantasma tendrán direcciones mayores que 25, por lo tanto, un paso después el nodo B puede transmitir el paquete con destino 10. Si un paquete fantasma llega a un nodo que contiene más de un paquete o bien un paquete que ha de esperar, entonces el paquete fantasma puede ser eliminado de la red.

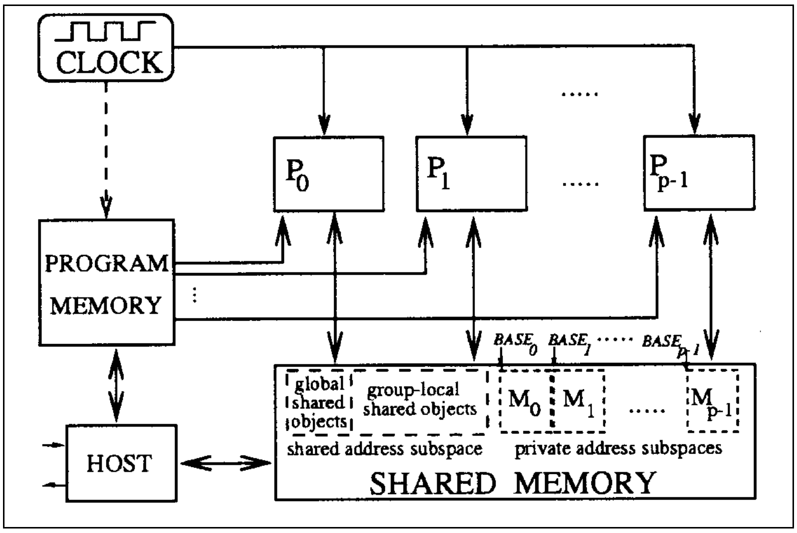

Figura 3.7 Estructura de la SB-PRAM desde el punto de vista del programador

3.3. Fork95

Fork95 [For], [Kes95a], [Kes95b], [Kes97a], es un lenguaje experimental paralelo imperativo fuertemente síncrono diseñado para expresar algoritmos PRAM. El lenguaje está basado en ANSI C [Ans90] y ofrece constructos para dividir jerárquicamente los grupos de procesadores en subgrupos y manipular subespacios de memoria privada y compartida. Fork95 es el sucesor de FORK [Hag92], [Kes94], que fue fundamentalmente un diseño de tipo teórico y del que nunca llegó a existir un compilador. En FORK los punteros, arrays dinámicos, estructuras de datos complejas y las sentencias de control de flujo no estructuradas fueron sacrificadas en aras de facilitar el diseño y comprobar la corrección del lenguaje. Estas limitaciones acabaron convirtiendo a FORK en un lenguaje sin posibilidad alguna de utilización práctica, no obstante, en FORK aparecían ya las ideas en las que se basa Fork95 como son el concepto de grupo de procesadores y la diferenciación entre espacios de memoria privados y compartido. En la actualidad, los autores de Fork95 trabajan en una nueva versión del lenguaje, que se llamará ForkLight y que pretenden que sea más adecuado para máquinas asíncronas de memoria compartida.

En Fork95 están presentes todas las características habituales en los lenguajes de programación secuencial. El lenguaje es en este caso un superconjunto de C, y las partes secuenciales son idénticas a las escritas en ese lenguaje. Aparte de librarle de las carencias que impedían una implementación útil de FORK, las mayores novedades de Fork95 con respecto a su antecesor es la introducción de cómputos localmente asíncronos para evitar los puntos de sincronización y permitir una mayor libertad a la hora de elegir el modelo de programación. Por otra parte, se ha abandonado la virtualización de procesadores limitando el número de procesos a los recursos hardware, y se han introducido punteros en el lenguaje.

Tanto FORK como Fork95 están muy ligados en su génesis al proyecto de construcción de la SB-PRAM. La Figura 3.7 presenta la estructura de la SB-PRAM desde el punto de vista del programador. Los subespacios de direcciones privadas de memoria de cada procesador están incrustados en la memoria compartida, y para su acceso se utilizan direcciones relativas a un puntero BASE para cada procesador. Todos los procesadores reciben la misma señal de reloj, de modo que la máquina es síncrona a nivel de instrucciones. El tiempo de acceso a memoria es uniforme para todos los procesadores y posiciones de memoria. El almacenamiento de una palabra de 32 bits en memoria consume un ciclo de CPU (es decir, lo mismo que una operación aritmética entera o en punto flotante), mientras que una lectura de un dato de este tipo consume dos ciclos de CPU. Como hemos estudiado, esta relación ideal de coste de comunicaciones/cómputo se alcanza gracias a la utilización de técnicas de hashing, ocultamiento de la latencia, combinación inteligente de nodos red y virtualización hardware de procesadores.

Fork95 proporciona dos modos de programación diferentes: el modo síncrono (que era el único disponible en FORK) y el asíncrono. Cada función se clasifica como síncrona o asíncrona. Dentro del modo síncrono los procesadores forman grupos que pueden ser divididos recursivamente en subgrupos, dando lugar a una estructura jerárquica de grupos en árbol. Las variables compartidas tienen asignada una memoria única para todo el grupo de procesadores que las creó, mientras que las variables privadas tienen asignada posiciones de memoria independientes en cada procesador. Todos los procesadores pertenecientes a un grupo hoja operan síncronamente. En el modo asíncrono, la librería de ejecución de Fork95 proporciona diferentes rutinas para implementar varias clases de semáforos, barreras, bucles paralelos equilibrados y colas paralelas, estructuras todas que son de interés en la implementación de algoritmos asíncronos. Por defecto, el modo de ejecución de los programas paralelos es el asíncrono. Dado que el lenguaje está basado en C, se puede incluir en los programas Fork95 código C (secuencial) estándar con muy pocos cambios.

En las siguientes secciones presentaremos con mayor detalle algunas de las características más importantes de Fork95, así como ejemplos de implementación de programas y su ejecución.

```
#include <fork.h>
#include <io.h>


void main(void) {
  if (__PROC_NR__ == 0)
    printf("Programa ejecutado por %d
procesadores\n",_STARTED_PROCS__);
  barrier;
  prS("Hola\n");
  barrier;
}
```

Figura 3.8 Un primer programa simple en Fork95

3.3.1. La ejecución de programas Fork95

Dado que la SB-PRAM, máquina objeto del compilador Fork95 está aún a nivel de pre-prototipo en periodo de pruebas, los miembros del equipo de trabajo Fork95/SB-PRAM han diseñado un simulador de la SB-PRAM con el que se pueden comprobar los algoritmos diseñados con el lenguaje. A este simulador se le ha llamado *pramsim* y con él hemos ejecutado todos los códigos fork95 que se presentan en esta memoria. El compilador de Fork95, *fcc* [Kes98] versión 1.7 y el simulador de la SB-PRAM se ejecutan en estaciones de trabajo SUN bajo SunOS o Solaris. La simulación de un gran número de procesadores requiere bastante memoria principal. El compilador de Fork95, *fcc*, genera código para el simulador de la SB-PRAM y su diseño está basado en el del compilador lcc 1.9 [Fra91], [Fra95], un compilador de ANSI C de una pasada. El compilador genera código ensamblador (ficheros *.s) que luego se procesan para convertirlos en formato COFF (ficheros *.o). El cargador de enlaces *plink* produce

```
***** PRAM-SIMULATOR (v2.0) *****
          >>>>>>>> DAMN FAST <<<<<<<<
        (c) by hirbli & stefran  07.10.94
You have 2 physical processors with 2 vP's each

Relocating file hola... done
Loading file /var/tmp/aaaa001pd.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Programa ejecutado por 4 procesadores
Hola
Hola
Hola
Hola
EXIT: vp=#0, pc=$000001fc
EXIT: vp=#1, pc=$000001fc
EXIT: vp=#2, pc=$000001fc
EXIT: vp=#3, pc=$000001fc
Stop nach 12194 Runden, 812.933 kIps
01fc 18137FFF  POPNG   R6, fffffff, R1
PRAM P0 = (p0, v0)> q
```

Figura 3.9 La ejecución del programa de la Figura 3.8

código ejecutable que correría en la SB-PRAM y también en el simulador.

La Figura 3.8 presenta un primer ejemplo de programa Fork95. Todos los programas han de incluir el fichero *fork.h*, que contiene las cabeceras de las rutinas que soportan paralelismo así como algunas macros. Como es habitual, el fichero *io.h* contiene las declaraciones de las funciones de entrada salida como *printf()* o la macro *prS()* que permite imprimir en pantalla una cadena de caracteres. Si este programa se graba con el nombre *hola.c* y se compila:

```
fcc –FORK –m –I$FORKDIR/include hola.c
```

y se ejecuta mediante el simulador *pramsim* se obtiene el resultado que aparece en la Figura 3.9, que indica al final que la ejecución del programa consumió 12194 ciclos (eso significa que cada procesador virtual de la SB-PRAM ejecutaría 12194 instrucciones, y la ejecución se llevó a cabo a 812.933 kIps (kilo instrucciones por segundo), lo cual es una medida de la rapidez de ejecución del intérprete. Los autores del simulador utilizan 'rounds' en lugar de ciclos de cpu porque en la implementación de la SB-PRAM, cada procesador físico soporta un máximo de 32 procesadores virtuales. El comando 'g' que aparece en la Figura 3.9 después del prompt del simulador es el que se utiliza para iniciar la ejecución del programa que se ha cargado. Si se escribe 'help' aparece una lista completa de los comandos disponibles que permiten, entre otras cosas, cambiar el número de procesadores (físicos y virtuales) que se van a utilizar por defecto. En nuestro ejemplo, la ejecución la llevaron a cabo cuatro procesadores virtuales, ejecutados por dos físicos, como se indica al comienzo de la ejecución.

```
#include <fork.h>
#include <io.h>

#define N 10
sh int sq[N];
sh int p;

void main(void) {
  pr int i;

  p = __STARTED_PROCS__;
  for (i = __PROC_NR__; i < N; i += p)
    sq[i] = i * i;
  barrier;
  for (i = __PROC_NR__; i < N; i += p)
    pprintf("i: %d sq: %d\n", i, sq[i]);
  barrier;
}
```

Figura 3.10 Un bucle de paralelismo de datos para calcular los cuadrados de N números

En Fork95 el programa de usuario (la función main()) es ejecutada por todos los procesadores disponibles en la PRAM en la que el programa haya sido arrancado. Este número de procesadores arrancados puede conocerse en tiempo de ejecución a través de la constante global entera y compartida *__STARTED_PROCS__*. Cada procesador mantiene una constante entera global y privada *__PROC_NR__* que es el identificador del procesador en tiempo de ejecución. Los procesadores se numeran consecutivamente desde 0 a *__STARTED_PROCS__-1*. Utilizando estos identificadores se puede escribir el típico bucle de paralelismo de datos como se muestra en la Figura 3.10 en la que el bucle se utiliza para calcular los cuadrados de N números. El programador siempre puede, a través del preprocesador utilizar nombres diferentes de *__STARTED_PROCS__* o *__PROC_NR__* para estos identificadores. La función *pprintf()* que aparece en la Figura 3.10 es completamente análoga al clásico *printf()* de C, con la salvedad que imprime en pantalla el identificador (*__PROC_NR__*) del procesador que la ejecuta, de forma que el resultado es el que aparece en la Figura 3.11. La sentencia *barrier*, que aparece en los dos programas que hemos presentado provoca una sincronización por barrera de todos los procesadores que están ejecutando el programa, de forma que todos han de alcanzar ese punto del código para poder proseguir la ejecución.

```
#0000# i: 0 sq: 0
#0001# i: 1 sq: 1
#0002# i: 2 sq: 4
#0003# i: 3 sq: 9
#0000# i: 4 sq: 16
#0001# i: 5 sq: 25
#0003# i: 7 sq: 49
#0002# i: 6 sq: 36
#0001# i: 9 sq: 81
#0000# i: 8 sq: 64
```

Figura 3.11 Resultado de la ejecución del programa de la Figura 3.10

3.3.2. Variables privadas y compartidas

Como hemos visto en la Figura 3.7, la memoria compartida de la PRAM se particiona, de acuerdo a lo que el usuario pretenda, en un espacio de direcciones privadas para cada procesador y un espacio de direcciones compartidas, comunes a todos los procesadores que podrá ser a su vez dinámicamente subdividido entre los diferentes grupos de procesadores. De acuerdo a esta división, las variables se clasifican en Fork95 como privadas (se han de declarar con el cualificador *pr*) o compartidas (*sh*). Por defecto, si una variable se declara sin cualificador de compartición, se asume que es privada. El término compartida se refiere en todo momento al grupo de procesadores que ha definido la variable. Definiremos más adelante con precisión el concepto de grupo de procesadores. Por ahora es suficiente saber que una variable compartida es accesible por todos los procesadores del grupo que la ha declarado y tiene asignada una posición única en memoria.

No es parte del lenguaje el definir lo que ocurre si varios procesadores escriben concurrentemente la misma variable compartida. Fork95 adopta el método de resolución de conflictos de la máquina objeto. En el caso de la SB-PRAM, triunfará el procesador con mayor *__PROC_NR__* (es decir, se trata de una *priority* CRCW-PRAM). De cualquier modo, dado que pueden utilizarse otros esquemas de resolución de conflictos de escritura, un buen programa no debería depender de estos esquemas, sino que debería utilizar elementos del propio lenguaje (como las operaciones de prefijos) para definir el comportamiento deseado.

IDENTIFICADOR	SIGNIFICADO	TIPO	COMENTARIO
`__STARTED_PROCS__`	Número de procesadores disponibles	**`sh int`**	sólo de lectura
`__PROC_NR__`	Identificador físico del procesador	**`pr int`**	sólo de lectura
`@`	Identificador lógico del grupo	**`sh int`**	puede redefinirse
`$`	Identificador lógico del procesador	**`pr int`**	puede redefinirse

Tabla 3.1 Identificadores especiales en Fork95

El resultado de una función que devuelva algún resultado será siempre privado. Obviamente, un valor retornado por una función puede hacerse accesible a todos los procesadores asignándolo a una variable compartida.

La variable especial **$** mantiene en cada momento el identificador lógico de cada procesador relativo a su grupo (en FORK este identificador se representaba por *#*). Inicialmente, *$* toma el valor del identificador físico del procesador (*__PROC_NR__*), pero puede cambiarse durante la ejecución del programa. La Tabla 3.1 presenta los identificadores especiales de Fork95.

La Tabla 3.2 resume la utilización de los cualificadores para el tipo de almacenamiento (privado o compartido de Fork95.

CUALIFICADOR	SIGNIFICADO	NOTA	EJEMPLO DE USO
`sh`	Declara una variable compartida		`sh int a=7, *p, v[10];`
`pr`	Declara una variable privada	defecto	`pr int i, *q, s[2];`

Tabla 3.2 Cualificadores de tipo de almacenamiento

3.3.3. Operaciones de prefijos

La SB-PRAM proporciona a nivel de instrucciones máquina operaciones de prefijos para las operaciones de suma, máximo, and y or lógicas que se realizan en dos ciclos de CPU para un máximo de 4096 procesadores físicos. Fork95 proporciona al programador estas operaciones como operadores atómicos para expresiones (no como funciones). Consideremos por ejemplo, la expresión

```
k = mpadd(ps, expresion);
```

Supongamos que esta sentencia es ejecutada simultáneamente por un conjunto P de procesadores. *ps* deberá ser un puntero (puede ser privado) a una variable entera compartida, s, y *expresion* ha de ser una expresión entera. Así pues, diferentes procesadores pueden apuntar a diferentes variables compartidas, s (usando punteros privados) de forma que diferentes sumas de prefijos se computen simultáneamente. Sea $Q_s \subseteq P$ el subconjunto de procesadores de P cuyos punteros apuntan a la misma dirección compartida s. Supongamos que los procesadores $q_{s,i} \in Q_s$ están indexados en orden creciente de su identificador físico, *__PROC_NR__* (este orden es heredado del hardware y no depende del identificador lógico, *$*). En primer lugar cada procesador $q_{s,i}$ de cada conjunto Q_s evalúa la expresión localmente dando lugar a un valor entero privado, $e_{s,i}$. Para cada posición de memoria compartida s direccionada, sea v_s el valor de s antes de la ejecución de *mpadd*. La instrucción *mpadd* computa simultáneamente para cada procesador $q_{s,i} \in Q_s$ el valor entero privado $v_s + e_{s,0} + e_{s,1} + \ldots + e_{s,i-1}$ y en el ejemplo considerado, este valor es asignado a la variable k. Inmediatamente después de la ejecución de *mpadd*, cada dirección de memoria compartida s contiene la suma de prefijos:

$$v_s + \sum_{j \in Q_s} e_{s,j}$$

de todas las expresiones participantes.

Si se ignora el valor devuelto por la instrucción mpadd(), ésta puede utilizarse para realizar una reducción global. Sin embargo, si se está interesado solamente en este efecto lateral se pueden utilizar las funciones *syncadd*, *syncmax*, *syncand* y *syncor* que llevan los mismos parámetros pero no retornan ningún valor. Por ejemplo, el número de procesadores disponibles puede calcularse en la variable *p* ejecutando síncronamente el código:

```
sh int p = 0;
syncadd(&p, 1);
```

La misma idea puede utilizarse para conseguir una renumeración consecutiva de los procesadores *0, 1, ..., p-1* almacenada en un identificador privado *name*:

```
sh int p = 0;
pr int name = mpadd(&p, 1);
```

```
 1 #define LIBRE 0
 2 #define BLOQUEADO   1
 3
 4 sh int lock = LIBRE;
 5
 6 void main(void) {
 7   if (__PROC_NR__ == 0)
 8     printf("%d Procesadores ejecutando\n", __STARTED_PROCS__);
 9   barrier;
10   printf("Procesador %d\n", __PROC_NR__);
11   while (mpadd(&lock, BLOQUEADO) != LIBRE)
12     ;
13   printf("Hola del procesador %d\n", __PROC_NR__);
14   lock = LIBRE;
15   barrier;
16 }
```

Figura 3.12 Una implementación simple del acceso a una sección crítica

La instrucción *mpadd* que se ejecuta cuando se traduce una expresión *mpadd()* se ejecuta atómicamente. En la SB-PRAM consume 2 ciclos de CPU. Por ello, esta instrucción (y el resto de operaciones de prefijos) pueden utilizarse para implementar el acceso exclusivo a secciones críticas de código como en el ejemplo que presentamos en la Figura 3.12.

```
PRAM P0 = (p0, v0)> g
4 Procesadores ejecutando
PPPPrrrroooocccceeeessssaaaaddddoooorrrr    1230



Hola del procesador 1
Hola del procesador 2
Hola del procesador 3
Hola del procesador 0
Stop nach 27152 Runden, 905.067 kIps
```

Figura 3.13 El programa de la Figura 3.12 ejecutado por 4 procesadores

En el programa de la Figura 3.12, el recurso compartido es la pantalla, y hemos de impedir el acceso concurrente a la misma, si queremos que las escrituras en pantalla no se entremezclen. De hecho, la escritura concurrente en pantalla que todos los procesadores que ejecutan este programa producen al alcanzar la llamada a *printf()* de la línea 10, provoca que estas escrituras aparezcan entremezcladas unas con otras, tal como muestra la Figura 3.13. En cambio, las escrituras producidas por la llamada a *printf()* de la línea 13 aparecen separadas en pantalla porque el acceso a pantalla ha sido convenientemente protegido mediante el uso de un 'cerrojo' (lock). Los procesadores producen su escritura según el número de procesador 1, 2, 3, 0, es decir, se considera el último procesador al que tiene *_PROC_NR_=0*.

Cuando la variable *lock* tiene un valor no nulo, la expresión *mpadd()* se evalúa a un valor no nulo en todos los procesadores, de modo que los procesadores ejecutan el bucle vacío de espera de las líneas 11 y 12 hasta que obtienen un valor cero de la expresión *mpadd()* (lo cual ocurre cuando se convierten en $q_{l,0}$ en la notación que utilizamos anteriormente). Puesto que el resto de procesadores que ejecutan el bucle de espera (los $q_{l,i}$ con i>0) obtienen valores no nulos, se garantiza que sólo un procesador accede a la sección crítica. Fork95 proporciona diferentes tipos de cerrojos que se implementan en ensamblador porque se utilizan muy frecuentemente en cómputos asíncronos. El código de la Figura 3.19 ilustra la utilización en Fork95 de un cerrojo simple mediante el tipo de datos *simple_lock*. Las funciones *simple_lock_init()*, que inicializa el cerrojo así como *simple_lockup()* y *simple_unlock()* que enmarcan la sección crítica, se definen en el fichero *fork.h* y están implementadas en ensamblador.

Fork95 soporta también los operadores *mpmax()* (máximo), *mpand()* (y lógica) y *mpor()* (o lógica). Estas operaciones están disponibles sólo para enteros debido a limitaciones hardware de la SB-PRAM.

Muchos algoritmos paralelos pueden implementarse elegantemente utilizando como bloques básicos las operaciones de prefijo [Ble89], de ahí la importancia de una eficiente implementación de las mismas. La librería PAD [Kes96] proporciona rutinas multiprefijo para tipos de datos arbitrarios con una complejidad temporal $O((n/p) + \log p)$, siendo p el número de procesadores y n el tamaño de los datos.

3.3.4. Zonas síncronas y asíncronas en los programas Fork95

Fork95 proporciona dos modos diferentes de programación asociados estáticamente (en tiempo de compilación) con zonas de código: el modo síncrono y el asíncrono. En el modo síncrono, los procesadores se mantienen sincronizados a nivel de sentencia, es decir, todos los procesadores ejecutan la misma sentencia en el mismo instante de tiempo y cumplen el que los autores de Fork95 llaman invariante de sincronicidad, que indica que los contadores de programa son iguales en cada instante de tiempo para todos los procesadores pertenecientes al mismo grupo activo. En modo asíncrono esto no tiene porqué ser cierto. En modo asíncrono la estructura del grupo de procesadores no puede alterarse (es de sólo lectura), no pueden alojarse (declararse) variables compartidas ni se puede alojar objetos en el heap compartido. En modo asíncrono no existen puntos de sincronización implícito entre los procesadores, pero pueden provocarse utilizando la sentencia *barrier*.

Al comienzo de un programa Fork95, todos los procesadores en los que el programa ha sido arrancado por el usuario ejecutan en paralelo el código de inicialización. A continuación, todos los procesadores comienzan la ejecución del programa en modo asíncrono invocando a la función *main()*.

Las funciones se clasifican en síncronas o asíncronas, y por defecto son asíncronas (ver Tabla 3.3). La función *main()* se considera asíncrona por defecto.

CUALIFICADOR	SIGNIFICADO	NOTA	EJEMPLO DE USO
`sync`	Declara (puntero a) función síncrona		`sync int tonta(sh int a) {...}`
`async`	Declara (puntero a) función asíncrona	defecto	`sh void(*tarea[a])(void);`

Tabla 3.3 Cualificadores de tipo de funciones y punteros

Una función síncrona se ejecuta en este modo (síncrono), excepto los bloques que comiencen con una sentencia *farm*:

```
farm <sentencia>
```

La sentencia *farm* provoca la entrada en modo asíncrono para la ejecución de *<sentencia>* y restablece el modo síncrono al final de su ejecución mediante una sincronización por barrera de todos los procesadores que han ejecutado el cuerpo.

La funciones asíncronas se ejecutan en este modo excepto los bloques que comiencen con una sentencia *start*:

```
start <sentencia>
```

La sentencia *start*, sólo se permite en modo asíncrono y cambia el modo de ejecución a síncrono para la sentencia que constituye su cuerpo. Provoca que todos los procesadores disponibles se sincronicen y ejecuten la sentencia simultáneamente en modo síncrono con identificadores lógicos *$* únicos numerados consecutivamente entre 0 y *__STARTED_PROCS__-1*.

Para poder mantener esta clasificación estática del código en síncrono o asíncrono, desde dentro de una zona asíncrona sólo se pueden invocar funciones asíncronas. Como contrapartida, si se invoca una función asíncrona desde una zona síncrona de programa, se provoca una entrada implícita en modo asíncrono. Es superfluo utilizar una sentencia farm desde un contexto asíncrono y ello puede provocar un bloqueo (el compilador genera un aviso).

Las funciones asíncronas no deben alojar variables locales compartidas. Esto implica que están prohibidos los parámetros formales compartidos para las funciones asíncronas. La implementación actual del compilador de Fork95 sólo permite sentencias start en el nivel más externo del programa. No se permite el anidamiento de sentencias start, lo cual es una debilidad del lenguaje que los autores de Fork95 tratan de obviar mediante una generalización de la sentencia start (la sentencia *join*) que permite el anidamiento arbitrario de zonas de código síncronas y asíncronas [Kes97].

La sentencia *seq*:

```
seq <sentencia>
```

funciona de forma similar a la sentencia *farm*. Su cuerpo es una zona asíncrona y se ejecuta por tanto en modo asíncrono, pero al contrario que *farm*, es ejecutada únicamente por un sólo procesador elegido arbitrariamente de entre el grupo de procesadores que ejecuta la sentencia *seq*. La sentencia *seq* trata de conseguir una implementación eficiente por parte del back-end del compilador que podría mantener a todos los procesadores del grupo ocupados en la ejecución de <sentencia> mediante la

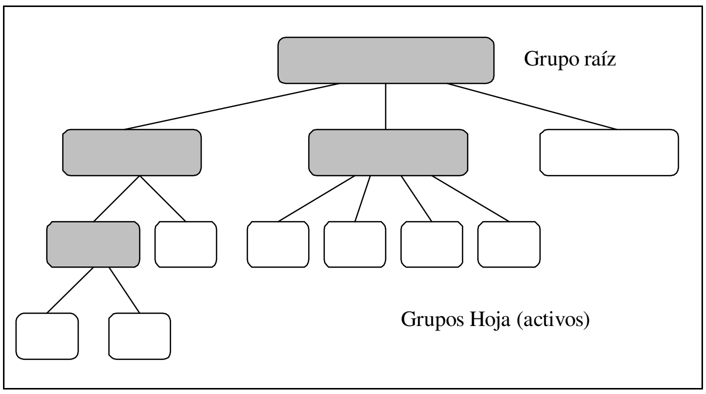

Figura 3.14 La jerarquía de grupos de Fork95

extracción de paralelismo a nivel de instrucción. El compilador emite un aviso si se utiliza una sentencia *seq* en una zona de código asíncrono.

3.3.5. El concepto de grupo en Fork95

Los programas Fork95 son ejecutados por grupos de procesadores, en lugar de como es habitual por procesadores individuales. Inicialmente hay sólo un grupo que contiene todos los procesadores disponibles y los grupos pueden ser subdivididos recursivamente. En cualquier punto de la ejecución del programa, todos los grupos existentes en ese momento constituyen una jerarquía de grupos que se estructura en forma de árbol (ver Figura 3.14).

Los subgrupos de un grupo dado se distinguen por su identificador de grupo. El identificador de grupo del grupo hoja al cual pertenece un procesador se accede a través de la variable compartida @. Inicialmente @ vale 0 pero su valor puede cambiarse de forma adecuada durante la ejecución del programa. Tanto el identificador de procesador $ como el de grupo, @ se guardan automáticamente cuando un grupo es dividido y se restablecen cuando el grupo se reactiva, no obstante, es responsabilidad del programador asignarle a estas variables valores adecuados.

En cualquier punto de la ejecución de un programa en modo síncrono Fork95 cumple el invariante de sincronicidad, de forma que todos los procesadores pertenecientes al mismo grupo de procesadores activos siguen el mismo flujo de control y ejecutan las mismas instrucciones en los mismos instantes. Además, todos los procesadores del grupo tienen acceso a un subespacio común de direcciones compartidas, de forma que las variables compartidas alojadas tienen asignadas una dirección de memoria única para cada grupo que las declara. Un procesador puede conocer el número de procesadores que pertenecen a su grupo a través de la rutina síncrona *groupsize()*.

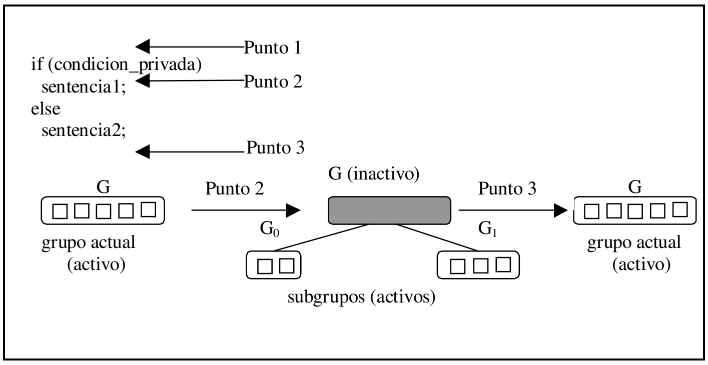

Figura 3.15 La división de los grupos en modo síncrono con una sentencia condicional con condición privada

A la entrada en un segmento de código síncrono, los procesadores forman un único grupo. No obstante, es posible que el flujo de control diverja en ramas cuyas condiciones dependan de valores privados. Para poder mantener el invariante de sincronicidad, el grupo hoja activo ha de dividirse en subgrupos. Se considera que una expresión es privada si no está garantizado que se evalúe al mismo valor en todos los procesadores (o lo que es lo mismo, si contiene una variable privada, una llamada a función o un operador multiprefijo). Las condiciones de sentencias condicionales o de bucle que sean compartidas no afectan a la sincronicidad, dado que la rama que se tome será la misma para todos los procesadores que la ejecuten.

Una condición privada en una sentencia condicional (if) provoca que el grupo actual de procesadores sea dividido en dos subgrupos (ver Figura 3.15): los procesadores para los que la condición se evaluó a un valor no nulo forman el primer subgrupo y ejecutan la rama correspondiente al *if*, mientras que los procesadores

```
 1 #include <fork.h>
 2 #include <io.h>
 3
 4 void main(void) {
 5   pr int i = $;
 6
 7   start {
 8     if ($ < 4) {
 9       @ = 23;
10       $ = 50-$;
11       if (i == 0)
12         farm pprintf("Rama if, $: %d @: %d\n", $, @);
13       else
14         farm pprintf("Rama else, $: %d @: %d\n", $, @);
15     }
16     else farm pprintf("No hace nada\n");
17   }
18 }
```

Figura 3.16 Un grupo inicial de procesadores se divide varias veces

restantes ejecutan la parte correspondiente al *else*. El grupo actual (padre) se desactiva y su espacio de direcciones compartidas disponible es dividido entre los nuevos subgrupos. Cada uno de los subgrupos establece una pila y un heap compartidos para su utilización para permitir que se declaren y alojen objetos compartidos relativos a cada subgrupo. Los subgrupos heredan sus identificadores de procesador y grupo (@ y *$*) del grupo padre, pero pueden redefinirlos localmente. Cuando ambos subgrupos han finalizado la ejecución de la sentencia condicional, se liberan y se reactiva el grupo padre mediante una sincronización explícita de todos sus procesadores. En el programa de la Figura 3.16, la sentencia condicional de la línea 8 divide el conjunto de procesadores que la ejecute en dos subconjuntos: aquellos con identificador menor que 4 y el resto. Los procesadores con identificador menor que 4 vuelven a ser divididos en dos subgrupos por la sentencia condicional de la línea 11, puesto que ésta también depende de una variable privada. La Figura 3.17 muestra el resultado de la ejecución del programa con 8 procesadores, y la Figura 3.18 muestra la jerarquía de grupos en el instante en que se ejecutan las dos alternativas correspondientes al *if* de la línea 11. En la Figura 3.18, los grupos que aparecen sombreados están inactivos en el instante de la computación que estamos considerando.

```
#0004# No hace nada
#0005# No hace nada
#0006# No hace nada
#0007# No hace nada
#0000# Rama if, $: 50 @: 23
#0001# Rama else, $: 49 @: 23
#0002# Rama else, $: 48 @: 23
#0003# Rama else, $: 47 @: 23
```

Figura 3.17 Ejecución del programa de la Figura 3.16

Un efecto similar tiene la ejecución de una sentencia de repetición si tiene condiciones privadas de salida. Supongamos que los procesadores de un grupo hoja, G alcanzan en modo síncrono una sentencia de repetición. Todos los procesadores que han de ejecutar la primera iteración de la sentencia formarán un grupo G' y se mantienen pertenecientes a G' mientras estén dentro del bucle de repetición. Una vez que la condición del bucle haya sido evaluada a cero por algún procesador de G', ese procesador abandona el grupo G' y espera al final de la sentencia de repetición para sincronizarse con todos los procesadores del grupo original, G. Dado este comportamiento, en la implementación de las sentencias de repetición no es necesario dividir la memoria compartida de G dado que los procesadores que abandonan la ejecución del bucle, simplemente esperan en una sincronización a que finalice el bucle el resto de procesadores.

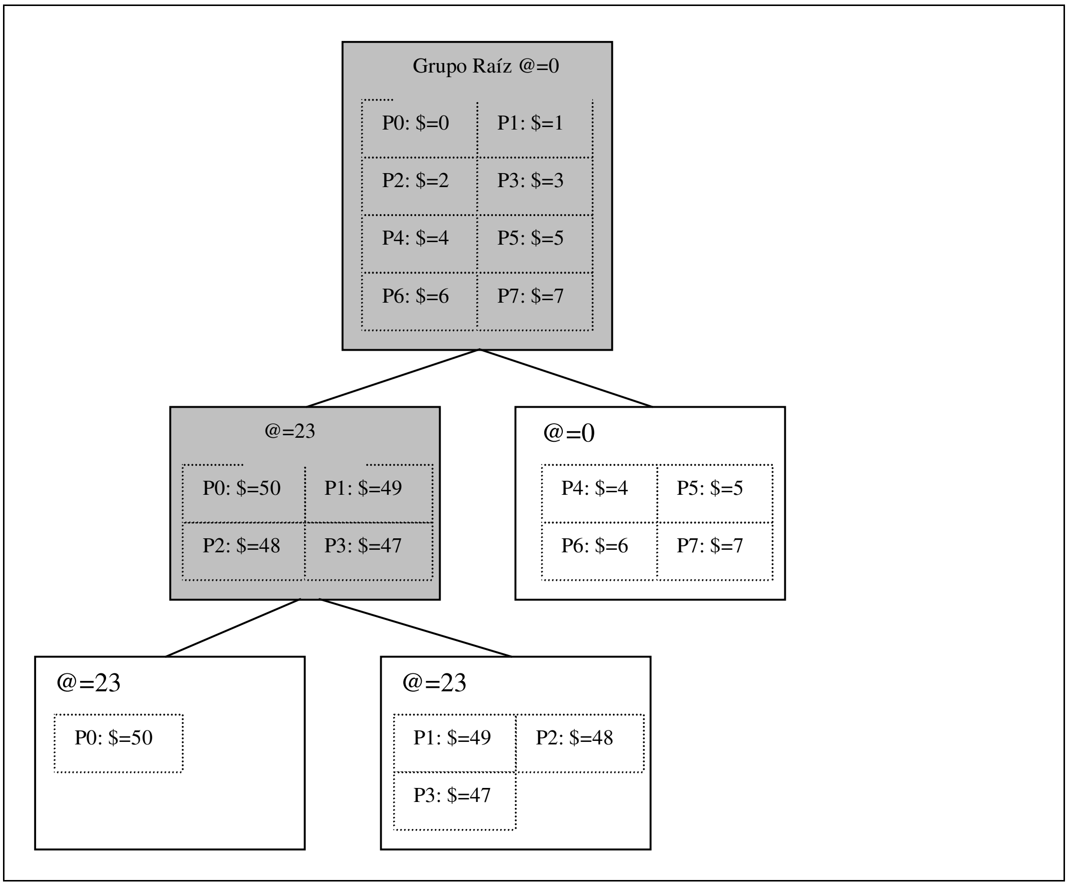

Figura 3.18 La jerarquía de grupos correspondiente a la ejecución del programa de la Figura 3.16

El siguiente código:

```
sh int p = groupsize();
pr int i;
for (i = $; i < n; i += p)
  a[i] += c * b[i];
```

ilustra la situación que hemos expuesto: los procesadores van abandonando el grupo que se crea para la ejecución del bucle for conforme finalizan la ejecución del bucle (cada procesador recorre un número diferente de iteraciones). Este típico bucle de paralelismo de datos se puede escribir como:

```
p = groupsize();
forall(i, 0, n, p) a[i] += c * b[i];
```

porque en el fichero *fork.h* está definida la macro:

```
#define forall(i,inf,sup,p)
    for (i=$+(inf); i<(sup); i+= (p))
```

que ejecuta su cuerpo con variable de bucle *i*, variando en el rango *[inf, sup)* utilizando todos los procesadores *p* del grupo hoja.

La división de grupos puede forzarse explícitamente en Fork95 mediante la sentencia fork (que da nombre al lenguaje). La ejecución de:

```
fork(e1; @=e2; $=e3) <sentencia>
```

significa lo siguiente: en primer lugar, se evalúa la expresión compartida e_1, que indica el número de subgrupos a crear, y el grupo hoja que ha ejecutado la sentencia se divide en e_1 subgrupos. Cada procesador evalúa e_2 para determinar el número del grupo hoja recién creado al que pertenece. El valor de e_3 indica para cada procesador el nuevo valor de su identificador dentro del subgrupo al que pertenece. El subespacio de memoria compartida del grupo padre se subdivide en tantas partes iguales como subgrupos se han creado y asigna una de ellas a cada subgrupo de forma que cada subgrupo disponga de su propia zona de memoria compartida. A continuación, cada subgrupo ejecuta *<sentencia>* de forma que los procesadores dentro de cada subgrupo ejecutan síncronamente, pero diferentes subgrupos pueden ejecutar diferentes flujos de control. Una vez que se finaliza la ejecución de *<sentencia>*, los procesadores de todos los subgrupos se sincronizan, las zonas de memoria compartida se mezclan, el grupo padre se reactiva como el grupo hoja en curso y todos los procesadores ejecutan síncronamente la siguiente sentencia que siga a la sentencia *fork*. Es posible que una sentencia *fork* cree subgrupos vacíos, aunque en ese caso, el trabajo de un grupo vacío finaliza inmediatamente. Al disponer de libre acceso a los nombres físicos en la sentencia *fork*, es el programador quien diseña las aplicaciones *Map* y *F* de asignación de tareas a procesadores que estudiamos en el Capítulo 2.

SENTENCIA	MODO	TIPO	PARÁMETROS	SIGNIFICADO
groupsize	sync	**int**	void	Retorna el número de procs. en el grupo
async_groupsize	async	**int**	void	Igual que groupsize pero para llamadas desde zonas asíncronas
parentgroupsize	sync	**int**	void	Retorna el número de procs. en el grupo padre.

Tabla 3.4 Inspección de la estructura de grupo

La Tabla 3.4 muestra las sentencias de Fork95 que permiten inspeccionar la estructura del grupo al que un procesador pertenece.

3.3.6. Ejecución síncrona y asíncrona. La sentencia `join`

El modo de ejecución síncrono elimina la necesidad de proteger variables compartidas mediante semáforos o cerrojos porque son accedidas de forma determinista: el programador puede confiar en un tiempo fijo de ejecución para cada operación, que es el mismo en todos los procesadores durante la ejecución del programa. No es necesario tomar precauciones adicionales para evitar comportamientos no deterministas. Si consideramos el código de la Figura 3.19 en el que la mitad de los procesadores ejecutarán la rama *if* de la sentencia condicional de la línea 9 y la otra mitad la rama del *else*, tenemos que no está determinado si la variable *y* acabará almacenando el valor antiguo o el nuevo de *a*, pero la semántica síncrona garantiza que todos los procesadores que escriben su variable privada *y* en la rama *else* del condicional le asignarán *el mismo* valor de *a*. No será necesario garantizar este hecho mediante una exclusión mutua.

El modo de ejecución estrictamente síncrono se corresponde con el estilo de programación PRAM, en la que el programador no ha de preocuparse por comportamientos no deterministas puesto que cada operación conlleva el mismo tiempo para todos los procesadores.

En las zonas de código asíncronas no existen puntos de sincronización implícitos. El mantener el invariante de sincronismo requiere un coste significativo, incluso para los casos en los que el grupo de procesadores que ejecuta el código contiene un único procesador, o cuando es innecesario el invariante de sincronismo porque no hay dependencias de datos. El convertir esas regiones de código en asíncronas conlleva un incremento sustancial en la velocidad de ejecución de los programas. Los autores de Fork95 indican que a través de la utilización de *farm* y funciones asíncronas consiguieron reducir los tiempos de ejecución (siempre medidos con el simulador *pramsim*) en un 50% [Kes97b].

La sentencia *join*, recientemente introducida en el lenguaje [Kes97b], es una generalización de la sentencia *start* que permite cambiar de modo de ejecución asíncrono a síncrono más fácilmente, así como el anidar arbitrariamente zonas de código síncronas y asíncronas. La sentencia tiene la siguiente sintaxis:

```
join(sentencia_retardo; condición_salida; SM_size)
  <sentencia>
else trabajo_util()
```

```
 1 sh int a = 1, x = 0;
 2 pr int y = 7;
 3 sh simple_lock lock;
 4
 5 void main(void) {
 6   simple_lock_init(lock);
 7   start {
 8     seq printf("%d Procesadores ejecutando\n", __STARTED_PROCS__);
 9     if (__PROC_NR__ % 2 == 0)
10       a = x;
11     else
12       y = a;
13   }
14   simple_lockup(&lock);
15   printf("Proc %d a: %d y: %d\n", __PROC_NR__, a, y);
16   simple_unlock(&lock);
17   barrier;
18 }
```

Figura 3.19 El código entre las líneas 9 y 15 es síncrono

La sentencia que aparece en el cuerpo de un *join* es una zona de código síncrono. En la sentencia *join*, varios procesadores se unen formando un grupo síncrono que ejecutan el cuerpo del *join*. Los procesadores que se unen en el *join*, pueden proceder de diferentes grupos hoja. El primer argumento de la sentencia *join*, la *sentencia_retardo* especifica una sentencia que es ejecutada por un procesador distinguido de los que forma el grupo del *join*, y modeliza una condición o un intervalo de tiempo que debe cumplirse para iniciar la ejecución de *sentencia*. La *condición_salida* es una expresión booleana suministrada por el programador, que puede ser diferente para diferentes procesadores y representa la condición que debe cumplirse para que un procesador abandone el grupo síncrono. El tercer parámetro de la sentencia, *SM_size* especifica el tamaño de memoria compartida que es alojada para disponer de una pila y heaps compartidos por los miembros del grupo que ejecutan el *join*. Esta memoria se libera cuando finaliza el cuerpo del *join*. La parte *else* de la sentencia *join* es opcional y especifica una sentencia asíncrona, *trabajo_util()* que es ejecutada por los procesadores que llegan tarde a la sentencia *join* (la ejecución del *join* ya se ha iniciado cuando un procesador la alcanza) y también por los procesadores que han abandonado el grupo del *join*. Una sentencia *retry* dentro del cuerpo de *trabajo_util()*, provoca que los procesadores que la ejecutan traten de ganar acceso al grupo síncrono del *join*. La sentencia *retry* funciona de forma similar a *continue* en los bucles.

En un programa no puede haber más de una sentencia *join*. Si el programador lo desea, puede encapsular la llamada dentro de una función e invocar la función desde diferentes puntos del código. A nivel de implementación hay una serie de variables de las que el programador podría hacer uso según sus intereses.

La semántica de la sentencia *join* se explica bien con una analogía que utiliza un autobús: los procesadores que se unen a un *join* son pasajeros que tratan de acceder a un autobús. Los pasajeros del autobús forman un grupo síncrono durante la ejecución del cuerpo del *join* (duración del trayecto). Los pasajeros han de acceder y salir del autobús en puntos concretos del recorrido (paradas del autobús), y si no lo consiguen, pueden volver a intentarlo la próxima ocasión que el autobús pase de nuevo por la parada (sentencia *retry*) pero mientras esperan, pueden también realizar algún trabajo alternativo (*trabajo_util()*).

SENTENCIA	SIGNIFICADO	EJEMPLO	MODO	CUERPO
`start`	Todos los procesadores entran en modo síncrono	**`start{v[$]=$;}`**	asíncrono	síncrono
`farm`	Entrada en modo asíncrono	**`farm puts("Hola");`**	síncrono	asíncrono
`fork`	Divide el grupo actual en subgrupos	**`fork(4;@=@%4;$=$/4)...`**	síncrono	síncrono
`seq`	Un procesador entra en modo asíncrono	**`seq printf(...);`**	síncrono	asíncrono
`join`	Forma un nuevo grupo raíz	**`join(wait();0;1000)...`**	asíncrono	síncrono
`retry`	Volver a intentar el 'join'	**`else{otra(); retry;}`**	asíncrono	-
`barrier`	Sincronización por barrera local al grupo	**`barrier;`**	asíncrono	-

Tabla 3.5 Sentencias

La implementación disponible actualmente de la sentencia **`join`** en Fork95 es una versión algo restringida que se realiza a través de una macro:

```
join(joinID, sentencia_retardo, condicion_salida,
     SM_size, reintentar, sentencia, trabajo_util);
```

en la macro, *reintentar* es un entero que indica si el procesador que ejecuta la macro ha de reintentar acceder al *join* si no lo consigue en primera instancia o bien si lo ha abandonado, y el primer argumento es un entero distinto para cada *join* que aparezca en el programa (en nuestra analogía sería el identificador del autobús).

Los autores utilizan las sentencia *join* para implementar lo que denominan secciones críticas síncronas paralelas como contraposición a las secciones críticas asíncronas secuenciales [Kes95b], no obstante no exploraremos en profundidad esta posibilidad dado que se aparta del modelo de programación PRAM.

La Tabla 3.5 resume las sentencias más importantes de Fork95 desde el punto de vista del paralelismo, indicando el modo de ejecución de la misma y su cuerpo, y un ejemplo de utilización de cada una.

3.3.7. Punteros y heaps

Al contrario que su antecesor, FORK, Fork95 ofrece al programador punteros. El uso de punteros es análogo al que de ellos hace C. No es preciso distinguir entre

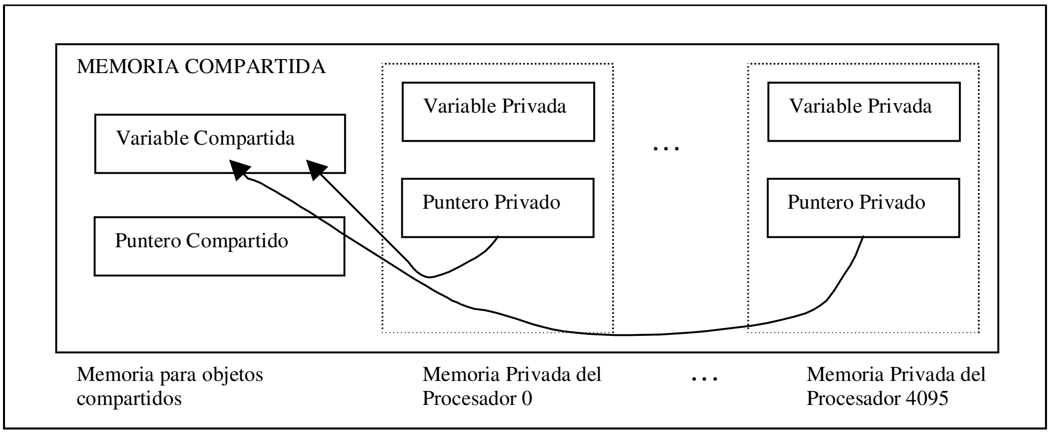

Figura 3.20 Diferentes punteros privados apuntando a la misma variable compartida

punteros a variables compartidas y privadas sino que las variables de tipo puntero compartidas pueden apuntar a variables privadas y viceversa. Es responsabilidad del programador establecer las relaciones convenientes en cada momento. La declaración

```
sh int *puntero_compartido;
```

corresponde a un puntero que puede apuntar a memoria compartida o privada.

El fragmento de código:

```
pr int privada, *puntero_privado;
sh int compartida;
puntero_privado = &compartida;
```

hace que las variables puntero (privados) *puntero_privado* de los procesadores que ejecuten la asignación apunten todas a una variable compartida (ver Figura 3.20). De forma análoga, si todos los procesadores ejecutan en paralelo la siguiente asignación:

```
puntero_compartido = &privada;
```

se produce una escritura concurrente, y de acuerdo a la semántica que hemos expuesto, la variable puntero (compartida) apuntará a la variable privada del procesador con mayor índice que participe en la asignación.

Las variables privadas pueden hacerse accesibles globalmente a través de los punteros, puesto que una variable puntero compartido que apunte a una variable privada hace que ésta sea accesible (a través del puntero) a todos los procesadores que declararon el puntero compartido (ver Figura 3.21).

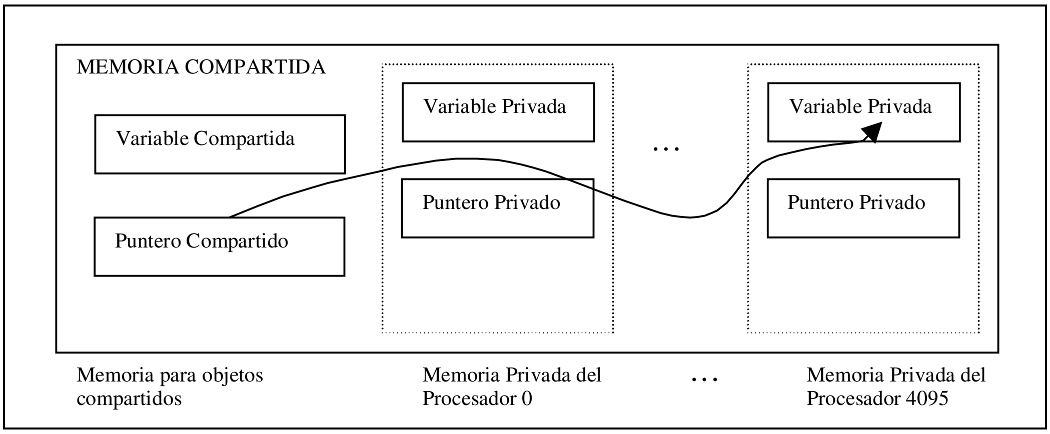

Figura 3.21 Una variable privada accesible a todos los procesadores a través de un puntero compartido

Fork95 dispone de tres tipos de heaps para el almacenamiento de objetos dinámicos: un heap privado para cada procesador, un heap compartido automático para cada grupo y un heap global compartido y permanente. Para alojar y liberar memoria en los heaps privados se utilizan las funciones (asíncronas) habituales de C, *malloc()* y *free()*. La memoria del heap compartido permanente se maneja utilizando las funciones asíncronas *shmalloc()* y *shfree()*. El heap automático compartido está pensado como una forma de suministrar memoria rápida y temporal a los procesadores de un grupo. El tiempo de vida de las entidades alojadas en el heap automático compartido mediante la función síncrona *shalloc()* está limitado al tiempo de vida del grupo de procesadores que ejecutó la función. Estas entidades se eliminan de la memoria si el grupo que las alojó finaliza su existencia. La función síncrona *shallfree()* libera todas las entidades dinámicas alojadas mediante *shalloc()* en la función (síncrona) que se esté ejecutando.

FUNCION	MODO	TIPO	PARÁMETROS	SIGNIFICADO
malloc **free**	**async** **async**	**char *** **void**	**pr uint** **pr char ***	Aloja memoria en el heap privado Libera memoria del heap privado
shmalloc **shfree**	**async** **async**	**char *** **void**	**pr uint** **pr char ***	Aloja memoria en el heap global compartido Libera memoria del heap global compartido
shalloc **shavail** **shallfree**	**sync** **async** **sync**	**char *** **uint** **void**	**shared uint** **void** **void**	Aloja memoria del heap compartido automático Tamaño libre del heap compartido automático Libera todos los bloques alojados con **shalloc**

Tabla 3.6 Rutinas para el manejo de memoria dinámica

Fork95 también permite punteros a funciones. Dado que a través de un puntero privado a función, cada procesador podría invocar síncronamente a una función diferente (y es imposible conocer estáticamente a cuál), las llamadas a funciones síncronas a través de un puntero privado son automáticamente convertidas por el compilador en llamadas asíncronas. Los punteros privados sólo pueden apuntar a funciones asíncronas.

La Tabla 3.6 resume las funciones de Fork95 para manejar los diferentes heaps disponibles. En esa tabla, *uint* es una abreviatura para *unsigned int*.

3.3.8. Precauciones a la hora de programar en Fork95

Si el programador coloca muchas sentencias condicionales anidadas con condiciones privadas, ello provocará sucesivas divisiones implícitas del grupo de procesadores hoja, con la consiguiente ineficiencia producida por el hecho de que hay que dividir la memoria compartida disponible entre los nuevos subgrupos creados.

La situación se hace especialmente crítica si las condiciones forman parte de una función que se invoca recursivamente: al cabo de unas pocas llamadas recursivas, la fragmentación que se produce en la memoria compartida hará que los bloques disponibles sean de un tamaño tan pequeño que a efectos prácticos, serán inservibles. Es responsabilidad del programador el evitar esta forma de programación.

Una buena práctica al programar en Fork95 si se pretende producir programas eficientes, consiste en conmutar a ejecución asíncrona desde el momento en que se sabe que todos los grupos hoja de la jerarquía de grupo contienen un único procesador. El pasar a modo asíncrono evita la sobrecarga asociada con la creación de subgrupos y la fragmentación de la memoria compartida.

```
#include <fork.h>
#include <io.h>

sync void output_array( sh int*, sh int );
sync void prefix_sum( sh int*, sh int, sh int *, sh int );

sh int *a, *b;
sh int n = 100;

sync void prefix_sum(sh int *in, sh int n, sh int *out,
                     sh int initsum) {
  pr int i;

  for (i=$; i<n; i+=p)
    out[i] = mpadd(&sum, in[i]);
}

void main(void) {
  pr int i;
  start {
    a = (int *) shalloc(n);
    b = (int *) shalloc(n);
    seq  prS("Vector inicial:\n");
    farm for (i=$; i<n; i+= __STARTED_PROCS__)
      a[i] = 1;
    output_array(a, n);
    prefix_sum(a, n, b, 0);
    seq  prS("Vector de salida:\n");
    output_array(b, n);
  }
}
```

Figura 3.22 La suma de prefijos en Fork95

3.3.9. Algunos ejemplos

3.3.9.1. La suma de prefijos

```
Relocating file prefix_sum... done
Loading file /var/tmp/aaaa002M1.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Vector inicial:
1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1

Vector de salida:
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
16  17  18  19  20  21  22  23 24

EXIT: vp=#0, pc=$00000377
EXIT: vp=#1, pc=$00000377
EXIT: vp=#2, pc=$00000377
EXIT: vp=#3, pc=$00000377
Stop nach 74366 Runden, 1239.433 kIps
0377 18137FFF  POPNG   R6, fffffff, R1
```

Figura 3.23 Ejecución del programa de la Figura 3.22

Basándonos en la instrucción mpadd(), la implementación de la suma de prefijos en Fork95 resulta tan trivial como el código que presentamos en la Figura 3.22. La función *main()* de la línea 18 aloja memoria del heap compartido para los vectores de entrada y de salida. El bucle *for* de la línea 24 inicializa el vector de entrada con un uno en cada componente mediante una ejecución asíncrona a través de la sentencia *farm*, e invoca en la línea 27 a la función *prefix_sum()*. La función toma como argumentos el vector de entrada, el número de componentes, el vector de salida, y un valor inicial, *initsum*, que se sumaría a todas las componentes del vector de salida (que en nuestro ejemplo inicializaremos a cero). La función divide el vector de entrada en N/p segmentos y el bucle de la línea 14 recorre todos los segmentos asignando un procesador a cada componente del segmento. Cada procesador, mediante la operación *mpadd()* de la línea 15 calcula la suma de prefijos de su componente en el vector de salida. El primer argumento de *mpadd()* va 'transmitiendo' la suma acumulada de cada uno de los segmentos hacia el siguiente. La función *prefix_sum()* supone que los identificadores *$* de los procesadores son números consecutivos en el rango [0..groupsize()-1]. En el código de la Figura 3.22 esto es cierto por el contexto en que la función es invocada, pero si no lo fuera, se podría conseguir fácilmente mediante otra llamada a *mpadd()*. La Figura 3.23 presenta el resultado de la ejecución del programa, utilizando un vector de 25 componentes y 4 procesadores.

3.3.9.2. El Quicksort

El programa que presentamos en la Figura 3.24 es una implementación en Fork95

```
sync void qs(sh int *array, sh int n , sh int p, sh int *temparray) {
sh int lowerprocs;           /* No. de procs. para los subgrupos */
sh int lowersize, uppersize; /* Tamaño de los vectores lower[], upper[] */
sh int equalsize;            /* No. de elementos iguales al pivot */
sh int *lower, *upper;       /* Subvectores a ordenar recursivamente */
sh int *equal, l, e, u;
sh int pivot = 0;
sh int aux = 0;
pr int j, k, mygroup;

  if ($ >= n) return;    /* Nunca se necesitan más de n procesadores */
  farm if ($ == 0) pprintf(" qs(%d,%d)\n", n, p);
  if (n <= 1)  return;                                   /* trivial */
  if (n == 2)  { sort2(array[0], array[1], pivot); return; }   /* simple */
  if (p == 1)  { farm qsort(array, n, 1, cmp); return; }       /* secuencial */
  renumber(aux);                               /* Poner $ con valores de 0 a p-1 */
  lowersize = uppersize = equalsize = 0;
  lowerprocs = 0;
  pivot = array[0];          /* Elegir un pivote */

  /* En paralelo, hallar los tamaños de los subvectores lower[], equal[], upper[]: */
  farm
    for (j = $; j < n; j += p) /* Recorrer el vector en paralelo */
      if (array[j] < pivot) mpadd(&lowersize, 1);
      else
      if (array[j] > pivot) mpadd(&uppersize, 1);
      else                  mpadd(&equalsize, 1);

  /* Asignar memoria a los subvectores lower[] y upper[] en temparray[]: */
  lower = temparray;
  equal = lower + lowersize;
  upper = equal + equalsize;

  /* En paralelo, copiar los elementos del vector hacia lower[]/equal[]/upper[] */
  l = e = u = 0;
  farm
    for (j = $; j < n; j += p)   /* Recorrer el vector en paralelo */
      if (array[j] < pivot)  { k = mpadd(&l, 1); lower[k] = array[j]; }
      else
      if (array[j] > pivot)  { k = mpadd(&u, 1); upper[k] = array[j]; }
      else                   { k = mpadd(&e, 1); equal[k] = array[j]; }

  farm
    for (j = $; j < n; j += p)
      array[j] = temparray[j];    /* Copiar de nuevo temparray en array */
  /* Ahora temparray[] se puede usar como temporal en llamadas recursivas */

  if (lowersize > 1 && uppersize > 1) {    /* Caso general */
    /* Subdividir los p procesadores en relación al tamaño de los subvectores */
    farm lowerprocs = (int)((float)(lowersize*p)/(float)(lowersize+uppersize));
    if (lowerprocs == 0) lowerprocs = 1;    /* corrección */
    if (lowerprocs == p) lowerprocs = p-1;  /* corrección */
    farm mygroup = ($ < lowerprocs) ? 0 : 1;
    fork (2; @=mygroup; $=$) {
      if (@ == 0) qs(array, lowersize, lowerprocs, temparray);
      else      qs(array + lowersize + equalsize,
                   uppersize, p - lowerprocs, temparray + lowersize + equalsize);
    }
  }
  else
  if (lowersize > 1)  qs(array, lowersize, p, temparray);
  else
  if (uppersize > 1)  qs(array + lowersize + equalsize,
                      uppersize, p, temparray + lowersize + equalsize);
  /* else nada que hacer; */
}
```

Figura 3.24 El Quicksort en Fork95

del algoritmo de ordenación Quicksort [Hoa61] que ya presentamos en el Capítulo 2. Para ordenar un vector, el algoritmo elige un elemento arbitrario del vector como elemento pivote y subdivide el vector en tres subvectores conteniendo los elementos menores, iguales y mayores que el pivote. A continuación procede recursivamente ordenando los subvectores que contienen los elementos menores y mayores que el pivote.

Estudiemos la implementación de la Figura 3.24 con más detalle: la función qs() es una función síncrona (línea 1) que toma como parámetros un puntero al vector a ordenar, *array*, el número de elementos del vector, *n*, el número de procesadores que se van a utilizar en la ordenación, *p* y un vector auxiliar, *temparray*, del mismo tamaño que el vector a ordenar. En el programa principal, la función se invocaría con *__STARTED_PROCS__* como tercer parámetro, indicando el número de procesadores a utilizar. En las líneas 13-15, el código trata los casos en que el número de elementos a ordenar es uno (en cuyo caso no hay nada que hacer), dos (en cuyo caso, si no están ordenados, se intercambian mediante una llamada a la macro *sort2* definida por el usuario) y el caso al que en general se debiera llegar en último término, en el que el número de procesadores a utilizar es uno, en cuyo caso se procede a ordenar el vector secuencialmente mediante una llamada a la función estándar *qsort* definida en *stdlib.h.* Si no nos encontramos ante ninguno de estos casos (es decir, hay más de un procesador disponible y el número de elementos es mayor que dos) el algoritmo asigna valores entre *0* y *p-1* a los *p* procesadores disponibles, mediante la llamada de la línea 16 a la macro definida por el usuario *renumber()* que utiliza una llamada a *mpadd()*. En la línea 19 se elige el primer elemento del vector a ordenar como elemento pivote (se podría elegir cualquier elemento) y en las líneas 22-27 el código halla en paralelo el número de elementos menores, iguales y mayores que el pivote elegido. Para ello se recorre todo el vector en bloques de tamaño igual al número de procesadores *p* utilizados asignando un procesador a cada posición del subvector considerado. Mediante las sentencias *mpadd()*de las líneas 24, 26 y 27 los procesadores incrementan en paralelo la variable correspondiente (*lowersize*, *uppersize* o *equalsize*) al caso del elemento considerado.

```
#include <fork.h>
#include <assert.h>
#include <io.h>
#define Nmax 8
sync void quicksort( sh int *);
sh int a[Nmax] = {4, 5, 7, 2, 3, 5, 6, 1};
pr int x, pos = 0;
main() {
   start (Nmax) {
      sh int numElem;
      x = a[$];
      quicksort( &numElemt );
      a[pos-1] = x;
   }
}
```

Figura 3.25 El programa principal para el Quicksort de la Figura 3.26

Una vez conocido el número de elementos menores, mayores e iguales al pivote, en las líneas 30-32 se asignan direcciones dentro del vector temporal *temparray* a los vectores *lower*, *equal* y *upper* que van a contener los elementos menores, iguales y mayores al pivote. Los elementos del vector *array* son copiados hacia su array correspondiente mediante el código de las líneas 35-41. Ese código utiliza una técnica similar: se recorre el array en paralelo mediante el típico bucle de paralelismo de datos y cada procesador se encarga de analizar una posición del vector fuente y de actualizar la correspondiente posición del vector de destino. La componente del vector de destino que ha de actualizar un determinado procesador se calcula mediante las sumas de prefijos que se calculan en las líneas 38, 40 y 41. El código de las líneas 43-45 copia el vector *temparray* de nuevo en su vector original, *array*. De este modo el vector temporal podrá ser reutilizado en las llamadas recursivas. El código de las líneas 48-59 trata el caso general en que hay elementos menores y mayores que el pivote (los casos particulares en que sólo hay elementos menores o sólo mayores se trata en el código de las líneas 61-64). En este caso general se divide el grupo de procesadores disponibles (recordemos que hay más de un procesador en este grupo) en dos subgrupos de forma que el número de procesadores asignados a cada subgrupo sea proporcional al tamaño del vector a ordenar. En el código de las líneas 50-52 se calcula el número de procesadores que se asignará al subgrupo que ordenará el subvector de menores. La llamada a *fork* de la línea 54 divide explícitamente en dos el grupo de procesadores, preservando para los nuevos grupos creados el identificador lógico de procesador y cambiando su identificador de grupo a 0 o 1 en función del identificador del procesador (línea 53). Los procesadores del grupo 0 ordenan recursivamente el vector de elementos menores que el pivote (línea 55) mientras que los del grupo 1 ordenan el vector de elementos mayores

```
 1  sync void quicksort( sh int *numElement) {
 2  sh int pivot, numLeft = 0, numRight = 0, totalPivots = 0;
 3  pr int left, right, pivotPos;
 4   pivot = x;
 5   left =  (x < pivot);
 6   right = (x > pivot);
 7   if (x == pivot) pivotPos  = mpadd(&totalPivots,1);
 8   else if (left)  quicksort( &numLeft );
 9   else quicksort( &numRight );
10   if (x == pivot)  pos = numLeft + pivotPos;
11   if (right) pos += totalPivots + numLeft;
12   *numElement = numLeft + numRight + totalPivots;
13  }
```

Figura 3.26 Otra implementación del Quicksort en *fork95*

que el pivote (líneas 56-57).

3.3.9.3. Otra implementación del Quicksort

Los códigos *fork95* de la Figura 3.25 y de la Figura 3.26 presentan otra elegante implementación del Quicksort en Fork95 basada en la creación implícita de grupos a través de la sentencia condicional. Cada uno de los procesadores copia mediante la asignación de la línea 11 (Figura 3.25) un elemento distinto del vector compartido *a* en la variable privada *x*. En la línea 4 de la Figura 3.26 los procesadores realizan una escritura concurrente en la variable compartida *pivot*. Uno de ellos, que podría ser elegido de manera no determinista, el programador no sabe cuál, logrará que su valor de *x = a[$]* sea escrito. Como consecuencia de la semántica de la sentencia condicional, las sentencias condicionales de las líneas 7, 8 y 9 dividen el grupo en 3 grupos. Uno formado por los procesadores cuya *x = a[$]* es menor que el pivote, otro formado con aquellos cuya *x* es mayor que el pivote y otro con los procesadores con *x = pivot*. Los dos primeros grupos proceden recursivamente a computar en la variable privada *pos* la posición del elemento *$* relativa al subvector ordenado. El tercer grupo realiza una suma de prefijos mediante la llamada a la función *mpadd*. Como consecuencia de esta llamada, en *pivotPos* queda la posición relativa del procesador dentro del grupo y en *totalPivots* el número de elementos en el subvector iguales al pivote. La posición relativa al subgrupo actual de los elementos iguales al pivote es computada en la línea 10. Para los elementos de la parte derecha es computada en la línea 11. Por último, la asignación de la línea 13 en la Figura 3.25 coloca cada elemento en la posición que le corresponde en el vector ordenado.

3.3.9.4. Mergesort

```
 1 sync void mergesort(int *array, int n, int *sortedarray) {
 2   sh int p = groupsize();
 3
 4   seq pprintf(" mergesort(array=%p, n=%d, p=%d)\n", array,n,p);
 5   if (n <= THRESHOLD || p == 1)
 6     seq seq_sort(array, n, sortedarray);
 7   else {
 8     sh int *temp = (int *)shalloc(n);
 9     fork (2; @= $ % 2; $ = $ / 2)
10        mergesort(array+@*(n/2),(1-@)*(n/2)+@*(n-n/2),temp+@*(n/2));
11     merge(temp, n/2, temp+n/2, n-n/2, sortedarray);
12   }
13 }
```

Figura 3.27 El mergesort en Fork95

La función *mergesort* que presentamos en la Figura 3.27 toma como parámetros un puntero a un vector de entrada, *array*, su tamaño, *n*, y otro puntero a un vector de salida ordenado, *sortedarray*. La función asume que la memoria para el vector de salida ha sido alojada antes de la llamada. La estrategia del programa consiste en dividir el

```
sync void merge(int *src1, int n1, int *src2, int n2, int *dest) {
  sh int p = groupsize();
  sh int iter;
  sh int *pos12, *pos21;  /* Vectores temporales */
  pr int i;

  farm assert(p > 1);
  pos12 = (int *)shalloc(n1);
  pos21 = (int *)shalloc(n2);
  iter = 0;
  farm
    for (i = mpadd(&iter, 1); i < n1; i = mpadd(&iter, 1))
      pos12[i] = get_pos(src1[i], src2, n2);
  iter = 0;
  farm
    for (i = mpadd(&iter, 1); i < n2; i = mpadd(&iter, 1))
      pos21[i] = get_pos(src2[i], src1, n1);
  farm {
    /* Copiar hacia dest usando la informaci¢n de posicion */
    for (i=$; i<n1; i+=p)  dest[i+pos12[i]] = src1[i];
    for (i=$; i<n2; i+=p)  dest[i+pos21[i]] = src2[i];
  }
}
```

Figura 3.28 La función merge

vector a ordenar en dos mitades de igual tamaño y ordenar recursivamente cada una de ellas dividiendo el grupo de procesadores disponibles en dos subgrupos (sentencia fork de la línea 9), cada uno de los cuales se encarga de una de las dos mitades. Ambos subgrupos de procesadores realizan la misma llamada recursiva a *mergesort*, pero utilizan datos de entrada y de salida diferentes porque los parámetros de la llamada se calculan en función de @, el identificador del grupo que hace la llamada.

Cuando el tamaño del vector a ordenar es suficientemente pequeño o bien el número de procesadores disponibles es sólo 1 (línea 5) se invoca a la función *seq_sort*, que ordena el vector secuencialmente. La función *merge* que se llama en la línea 11 se encarga de mezclar en paralelo los dos subvectores ordenados dejando el resultado almacenado en el vector *sortedarray*.

```
***** PRAM-SIMULATOR (v2.0) *****
         >>>>>>>> DAMN FAST <<<<<<<<
       (c) by hirbli & stefran  07.10.94
You have 5 physical processors with 1 vP's each

Relocating file mergesort... done
Loading file /var/tmp/baaa002o5.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Enter N = 32
#0000#  mergesort(array=620cb, n=32, p=5)
#0000#  mergesort(array=620cb, n=16, p=3)
#0001#  mergesort(array=620db, n=16, p=2)
#0000#  mergesort(array=620cb, n=8, p=2)
#0001#  mergesort(array=620db, n=8, p=1)
#0002#  mergesort(array=620d3, n=8, p=1)
#0003#  mergesort(array=620e3, n=8, p=1)
#0000#  mergesort(array=620cb, n=4, p=1)
#0004#  mergesort(array=620cf, n=4, p=1)
#0000# Array 620ab of size 32:
 -932 -812 -799 -691 -688 -656 -555 -539 -523 -437 -391 -366 -201 -182
-171 -151 -132 -51 -2 38 198 203 291 297 368 395 673 694 703 855 926
931
```

Figura 3.29 Resultado de la ejecución del mergesort con 5 procesadores

El código de la función *merge* aparece en la Figura 3.28. La función mezcla los dos vectores ordenados *src1* y *src2* de tamaños *n1* y *n2* respectivamente en el vector *dest*. La función utiliza dos vectores auxiliares, *pos12* y *pos21* para calcular en ellos la posición de cada elemento de cada uno de los dos vectores en el otro vector (*pos12[i]* almacenará el número de elementos de *src2* que son menores que el elemento *src1[i]*). A partir de esta información, los bucles de las líneas 20 y 21 de la Figura 3.28 colocan cada uno de los elementos de *src1* y *src2* en su posición definitiva en *dest*. El cálculo de los valores de los vectores *pos12* y *pos21* se realiza mediante una búsqueda binaria secuencial, a través de la función *get_pos()*. Los bucles de las líneas 12 y 16 utilizan la sentencia *mpadd()* para paralelizar el cómputo de los vectores pos. La inicialización e incremento de la variable del bucle hace que cada procesador calcule la posición de un elemento distinto.

La implementación que presentamos del mergesort asume que todos los elementos a ordenar son distintos y la Figura 3.29 presenta el resultado de la ejecución del programa utilizando 5 procesadores con un vector de 32 elementos generado aleatoriamente. Vemos que en la primera división del vector, el grupo de 5 procesadores se divide en dos subgrupos con 3 y 2 procesadores, y posteriormente el grupo con 3 procesadores se dividirá en dos subgrupos con 2 y 1. Los procesadores individuales finalmente ordenan secuencialmente segmentos del vector original de tamaños 8, 8, 8, 4 y 4.

3.3.9.5. La FFT

Dada una serie de números $\{a_0, a_1,..., a_{n-1}\}$ su transformada discreta de Fourier es la secuencia $\{b_0, b_1,..., b_{n-1}\}$ siendo

$$b_j = \sum_{k=0}^{n-1} a_k * w^{kj} \quad para \quad j = 0,1,\cdots n-1$$

$$siendo \quad w = e^{2\pi i/n} = \cos\left(\frac{2\pi}{n}\right) + i\ sin\left(\frac{2\pi}{n}\right)$$

$$i = \sqrt{-1}$$

Esto es, $w^n = e^{2\pi i} = 1$ y $1, w, w^2, w^3, ..., w^{n-1}$ coincide con las raíces n-ésimas de la unidad en el plano complejo. Si n es una potencia de dos, $n=2^s$ para algún entero s, la expresión para los b_j Se puede reescribir como:

$$b_j = \sum_{m=0}^{2^{s-1}-1} a_{2m}\ e^{2\pi ijm/2^{s-1}} + w^j \sum_{m=0}^{2^{s-1}-1} a_{2m+1}\ e^{2\pi ijm/2^{s-1}}$$

```
sync cplx *fft(sh cplx *a, sh int n, sh cplx *w) {
  sh cplx *ft;                            /* Vector para el resultado */
  sh cplx *even, *odd, *fteven, *ftodd;   /* Punteros temporales */
  sh int p = 0;
  sh int ndiv2;
  pr int i;

  $ = mpadd(&p, 1);  /* Asegurar numeración consecutiva de los procesadores */
  seq  prS("fft\n");
  if (n==1)  {
    seq {
      ft = shmalloc(1);
      ft[0] = cnum(a[0]->re, a[0]->im);
    }
    return ft;
  }
  if (p==1) return seq_fft(a, n, w);
  seq ft = (cplx *) shmalloc(n);  /* Memoria para el resultado */
  ndiv2 = n >> 1;
  even = (cplx *) shalloc(ndiv2);
  odd  = (cplx *) shalloc(ndiv2);
  for(i = $; i < ndiv2; i += p) { /* Bucle paralelo */
    even[i] = a[2 * i];           /* Cálculo de componentes pares e impares */
    odd[i]  = a[2 * i + 1];
  }
 if ($ < p/2)
   fteven = fft(even, ndiv2, w);
 else
   ftodd = fft(odd, ndiv2, w);
 farm
   for(i = $; i < ndiv2; i += p) {     /* Bucle paralelo */
     pr cplx t      = cmul(w[i], ftodd[i]);
     ft[i]          = cadd(fteven[i], t);
     ft[i + ndiv2] = csub(fteven[i], t);
     freecplx(t);
   }
 seq shfreecplxarray(fteven, ndiv2);
 seq shfreecplxarray(ftodd, ndiv2);
 shallfree();  /* Liberar los vectores even[] y odd[] */
 return ft;
}
```

Figura 3.30 La Transformada rápida de Fourier en Fork95

Al algoritmo que calcula la transformada discreta de Fourier mediante esta expresión se le conoce como transformada rápida de Fourier (FFT) [Akl89]. Se trata de un algoritmo recursivo, dado que en la expresión anterior, cada una de las sumas es a su vez una transformada discreta de Fourier. La primera suma corresponde a los términos pares y otra a los impares de la secuencia de entrada.

La FFT es un algoritmo de amplísimo uso en un gran espectro de campos, entre los que podemos reseñar el procesamiento digital de señales, teoría de la codificación, transmisión del habla, proceso de imágenes, predicción meteorológica, etc. De ahí la importancia del diseño de algoritmos rápidos para realizar este cómputo.

La función *fft* de la Figura 3.30 es solución recursiva para el cálculo de la transformada rápida de Fourier del vector de complejos *a* que recibe como primer parámetro. El segundo parámetro, *n* es el número de componentes del vector, que se asume que es una potencia de dos, mientras que el tercer parámetro es un vector que contiene las potencias de la raíz n-ésima de la unidad en el plano complejo, $1, w, w^2, .., w^{n-1}$ siendo $w = e^{2\pi i/n}$.

El código entre las líneas 9-15 trata el caso trivial en que el número de componentes del vector es uno. En la línea 16, mediante la llamada a la función *seq_fft()* se resuelve el caso en que el número de procesadores disponibles es sólo uno, en cuyo caso el problema se resuelve secuencialmente. El resto del código (líneas 17-40) resuelve el caso general en que el número de componentes y el número de procesadores disponibles es mayor que uno. En este caso el algoritmo procede recursivamente en paralelo dividiendo el vector original en sus componentes pares e impares (bucle paralelo de las líneas 21-24) que almacena en los vectores *even* y *odd*. A continuación el grupo de procesadores disponibles se divide en dos subgrupos de igual tamaño (sentencia condicional de las líneas 25-28) de forma que la mitad de los procesadores (aquellos con identificador lógico menor que la mitad del número de procesadores disponibles) llaman recursivamente a *fft()* sobre el subvector de componentes pares, mientras que el resto opera sobre las componentes impares. El bucle paralelo de las líneas 29-35 se encarga de la combinación de los resultados de la transformada de componentes pares e impares generando el vector de resultados. Las funciones *cadd(), cmul()* y *csub()* que aparecen en este bucle se encargan de realizar la suma, producto y sustracción de números complejos. La función finaliza liberando la memoria alojada para resultados temporales.

3.3.9.6. La Quickhull

La función *qh()* que presentamos en la Figura 3.32 calcula la envolvente convexa de una nube de puntos. Se trata de una versión paralela del algoritmo Quickhull [PRE85] para hallar la envoltura convexa de una nube de puntos. El algoritmo *quickhull* recibe este nombre por su gran parecido con el conocido algoritmo de ordenación Quicksort. Dados N puntos en el plano, el problema consiste en hallar cuales de estos puntos se encuentran en el perímetro de la menor región convexa que contenga a todos los puntos. Se trata de un problema que tiene multitud de aplicaciones, especialmente en el campo de la geometría computacional.

El conjunto de puntos al que se ha de calcular la envolvente convexa se puede dividir en dos subconjuntos de puntos: aquellos que quedan en cada semiplano de los definidos por una línea que une dos puntos con máxima distancia entre sí, *p1* y *p2* en la Figura 3.31. La función *qh()* se puede utilizar para calcular la envolvente de cada uno de los dos subconjuntos. El primer parámetro de la función, *pt[]* es un vector de índices al vector de puntos, en cuyas dos primeras componentes se encuentran los dos puntos *p1* y *p2* que definen el problema que se está resolviendo, y el segundo parámetro indica el número de puntos del vector *pt*.

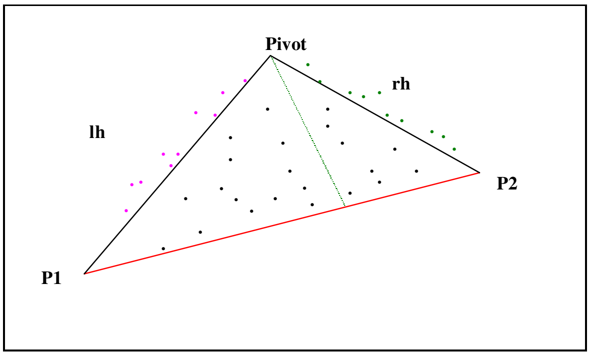

Figura 3.31 Cálculo de la envoltura convexa de una nube de puntos

En las líneas 10-15 del código de la Figura 3.32 se abordan los casos simples: si el número de puntos es dos, no hay nada que hacer, si son tres, hay que marcar el que se encuentra entre p1 y p2 como perteneciente a la envolvente y si el número de procesadores disponibles es sólo uno, el problema se resuelve mediante la función secuencial *seq_qh()* que se invoca en la línea 15..

```
 1 sync void qh(sh int *pt, sh int n) {
 2   sh int pivot;
 3   sh int p = 0;              /* Número de procesadores disponibles */
 4   sh int p1=pt[0], p2=pt[1];    /* Linea (p1, p2) */
 5   sh int *lh, *rh, n1, n2;     /* Para almacenar los subproblemas */
 6   sh int firstrightproc;
 7
 8   farm pprintf("qh(%d)\n", n);
 9   $ = mpadd(&p, 1);    /* Renumera $ y determina p */
10   if (n==2) return;
11   if (n==3) {   /* El punto entre p1 y p2 pertenece al hull */
12     belongs_to_hull[pt[2]] = 1;
13     return;
14   }
15   if (p==1) { seq_qh(pt, n); return; }
16
17  pivot = pivotize(pt, n, p);
18  belongs_to_hull[pivot] = 1;
19
20   /* Hallar subproblemas: */
21   n1 = n2 = n;
22   lh = delete_right(pt, &n1, p1, pivot);
23   rh = delete_right(pt, &n2, pivot, p2);
24
25   /* Determinar cantidad de procesadores en cada subproblema */
26   firstrightproc = (int) ((float)p * est_work(n1) / est_work(n));
27   if (firstrightproc == 0 && n1 > 0) firstrightproc = 1;
28   if ($ < firstrightproc)   /* Dividir el grupo de procesadores: */
29     qh(lh, n1);
30   else
31     qh(rh, n2);
32 }
```

Figura 3.32 La Quickhull en Fork95

En el caso general, el algoritmo elige un punto como pivote (línea 17) mediante la llamada a *pivotize()* y lo marca como perteneciente a la envolvente (línea 18). El vector global *belongs_to_hull[]* almacena los índices de los puntos pertenecientes a la envoltura convexa. Como pivote, la función *pivotize()* devuelve el punto de mayor distancia a la recta P1P2. Las llamadas a *delete_right()* de las líneas 22 y 23 se utilizan para descomponer el problema en dos subproblemas, que se almacenan en los vectores *lh* y *rh* y que corresponden a los puntos a la izquierda de la línea *P1-pivot* y a la derecha de la línea *Pivot-P2* (ver Figura 3.31). La función *delete_right()* devuelve un vector del que se han eliminado los puntos de *pt* que quedan a la derecha de los puntos indicados por sus dos últimos parámetros. A continuación, en las líneas 26 y 27 el programa calcula el número de procesadores que se dedicará a la solución de cada subproblema mediante una llamada a la macro *est_work()* definida por el usuario, que proporciona una estimación de la dificultad del problema en cuestión. El grupo de procesadores disponibles será dividido en dos subgrupos y el número de procesadores de cada grupo dependerá de esta estimación. En las líneas 29 y 31 del código, la función *qh()* es llamada recursivamente para resolver los dos subproblemas resultantes. El primer subgrupo de procesadores opera sobre *lh* y el segundo sobre *rh*.