Curso 1996/97
CIENCIAS Y TECNOLOGÍAS

COROMOTO LEÓN HERNÁNDEZ

Diseño e implementación de lenguajes orientados al modelo PRAM

Director
CASIANO RODRÍGUEZ LEÓN

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE LA LAGUNA

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

A Juana y Elicio

A Laly, Juany, Care y Blas Elicio

AGRADECIMIENTOS

En la realización de este trabajo han colaborado muchas personas, a todas ellas expreso mi gratitud.

Al director de mi trabajo, D. Casiano Rodríguez León, por su insustituible orientación y cooperación.

A mis compañeros del Departamento de Estadística, Investigación Operativa y Computación, especialmente a Kiko, José Luis, Félix, Paco, Dani, Patri y Javi, por su apoyo y amistad.

Al Centro Superior de Informática por poner a nuestra disposición todos sus recursos.

A mis padres por haberme brindado la oportunidad de aprender.

Índice

Prólogo

Capítulo I
El Modelo PRAM

Capítulo II

Lenguajes Orientados al Modelo PRAM

Capítulo III
El Lenguaje ll

Conclusiones y Trabajos Futuros

Apéndice A

Apéndice B

Bibliografía

Prólogo

El argumento más utilizado para mostrar las ventajas del procesamiento paralelo es su eventual mejora en el rendimiento. Si la única motivación para el paralelismo fuera incrementar el rendimiento de los sistemas de computación, el paralelismo se convertiría sólo en otra propuesta tecnológica y el usuario no tendría que conocer su existencia. En ese caso, los lenguajes de programación deberían aislar al programador del paralelismo, como esconden otros detalles de la arquitectura objeto (por ejemplo, el número y tipo de registros, los niveles y las jerarquías del sistema de memoria, la segmentación –*pipeline*– de los superescalares, etc.). Sin embargo, las posibilidades de éxito de esta política son escasas porque el paralelismo es una idea con entidad suficiente que no debería ocultarse a aquellos que tratan de resolver problemas pues aparece continuamente en los problemas del mundo real.

Cuando se plantea la resolución de un problema, se debe construir un modelo lo suficientemente detallado de sus partes más relevantes. Las metodologías de diseño de software reconocen el paralelismo natural, si

está presente, y lo formalizan en las primeras etapas del diseño. Sin embargo a continuación se encuentran con el considerable problema de eliminarlo para producir un modelo secuencial que se pueda expresar en un lenguaje de programación tradicional. Esta fase de secuencialización no es trivial y es una fuente de errores. El modelo secuencial resultante no tiene una correspondencia directa con el diseño paralelo original, incluso en sus componentes más simples, lo que complica y encarece las tareas de verificación, comprobación y mantenimiento del sistema. Por lo tanto, si se pudiera retener el paralelismo inherente en el problema original se evitarían las dificultades y el esfuerzo que supone la etapa de secuencialización. Para conseguir este objetivo, se tiene que permitir y estimular el uso del concepto de paralelismo entre los usuarios.

Sin embargo, si un usuario quiere abordar la resolución de un problema haciendo uso de paralelismo se le advierte que se encontrará con inconvenientes, tales como:

- La programación paralela es más compleja que la programación secuencial
- No se puede garantizar la obtención de resultados fiables, pues el procesamiento paralelo es inherentemente no determinista.
- El procesamiento paralelo introduce nuevos y complejos problemas para el diseñador, como por ejemplo los interbloqueos (*deadlock* y los *livelock*).
- Como consecuencia de lo anterior, es muy difícil depurar programas paralelos.
- Los programas paralelos son muy difíciles de transportar entre arquitecturas paralelas diferentes.

La resolución de estos inconvenientes se ve impedida por la inmadurez actual del procesamiento paralelo. Prueba de esta inmadurez es la gran competencia de paradigmas y tecnologías.

El paralelismo se encuentra por doquier desde el punto de vista hardware, existiendo una gran abundancia de alternativas tecnológicas. Si bien hay un acuerdo sobre los conceptos fundamentales a nivel de un único procesador, para multiprocesadores, las ideas divergen marcadamente; prueba de ello son los modelos SIMD, SPMD y MIMD. Los procesadores interactúan mediante una memoria compartida (protegida por semáforos, con sincronizaciones por barrera, coherencia hardware de la cache, etc.) y/o mediante el paso de mensajes (con o sin utilización de *buffers*, síncronos/asíncronos, punto-a-punto/*broadcast*/muchos-a-uno, etc.).

Por el lado software, se tienen paradigmas tales como el *paralelismo de datos* y el *paso de mensajes*. Se cuenta con métodos funcionales para la descomposición de procesos, principios clientes-servidor, sincronizaciones por barrera, granjas, llamadas a procedimiento remotas y rendezvous.

Actualmente el vacío entre el paralelismo hardware y los lenguajes de programación de alto nivel es sustancial, de ahí que la elección de una plataforma paralela hardware provoque serias restricciones sobre la elección del paradigma de software paralelo. La tecnología paralela elegida matiza de forma importante la complicación de los problemas con los que se puede encontrar el usuario. El resultado es que los programadores deben codificar a bajo nivel, entendiéndose este como un nivel orientado a la máquina para la que se escribe el algoritmo, con lo que los programas paralelos resultan poco o nada portables.

Este estado de las cosas no es sorprendente, dado que muchas de las variables involucradas en el diseño de un sistema paralelo están en un estado permanente de fluctuación. Estas variables incluyen las capacidades que las máquinas paralelas pueden ofrecer, las técnicas de traducción y optimización de los compiladores para máquinas paralelas y las construcciones de alto nivel apropiadas en los lenguajes de programación paralelos. En suma, los sistemas de arquitecturas paralelas permiten muchos más grados de libertad que los sistemas secuenciales. Sin embargo, la práctica de reescribir programas para cada nueva arquitectura paralela es económicamente inaceptable.

Así pues, el procesamiento paralelo precisa mejorar en varios aspectos. En primer lugar, es necesario facilitar el desarrollo de aplicaciones para el hardware paralelo, buscando una mayor simplicidad en su manipulación. Es un objetivo primordial el alcanzar una conciliación entre las arquitecturas de máquinas paralelas y los lenguajes de programación para permitir que los programas puedan escribirse en lenguajes de alto nivel orientados a los problemas. En segundo lugar podemos citar la necesidad de que los sistemas paralelos ofrezcan una mayor eficiencia. Por último, es esencial el desarrollo de aplicaciones portables para el hardware paralelo que sean independientes de características específicas.

Si se colocan la eficiencia y la portabilidad como la meta principal para la computación paralela se restringen los objetivos a la obtención de beneficios. La simplicidad es la meta más difícil de conseguir, pero la que ofrece más ventajas. La búsqueda de la simplicidad requiere, algunas veces, la reevaluación de las estrategias básicas. Para la computación en paralelo, la simplicidad impone una integración clara de los conceptos paralelos a todos los niveles, tanto en el diseño del sistema como en la implementación. Si se obtiene un buen sistema, la eficiencia y la portabilidad serán consecuencias naturales.

Los puntos de vista expuestos en los párrafos anteriores pueden aproximarse según Welch utilizando la siguiente estrategia. En primer lugar, se requiere que al desarrollar una aplicación se ponga al descubierto su naturaleza paralela, la cual se debe preservar explícitamente a través del diseño y de la implementación. Además, se debe resguardar la aplicación de la naturaleza paralela del hardware para el que se realiza el sistema. La integración del paralelismo debe formar parte de la sintaxis y la semántica del lenguaje de programación en contraposición al acceso mediante librerías en tiempo de ejecución. Si a un usuario se le da la facilidad de expresar paralelismo (a cualquier nivel de diseño e implementación) con la misma facilidad que (digamos) tiene al utilizar una sentencia condicional o un bucle, se permite que la expresión de una solución se convierta en algo

más cercano a la aplicación y se simplifican las tareas de verificación y mantenimiento. Pero el paralelismo inherente a tal expresión ha de ser en términos de la aplicación y no en términos del hardware objeto. Se tienen entonces dos tipo de paralelismo, y uno puede ser completamente diferente del otro, por ejemplo en la granularidad de los procesos. Permitir y aproximar esta división es uno de los problemas a los que se enfrenta la computación paralela.

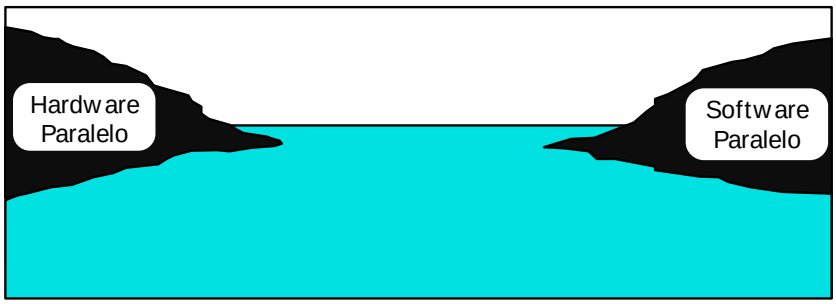


Figura 1

Valiant describe cómo construir un puente entre el hardware paralelo y el software paralelo que posibilita la aplicación eficiente de uno en el otro, a pesar de los modelos subyacentes en cada lado (Figura 1). Dicho autor, hace notar que este mismo problema ha sido resuelto en la computación secuencial. El concepto de máquina de von Newman ha hecho posible que estilos muy diferentes de hardware y software secuenciales se soporten mutuamente con gran eficiencia (Figura 2).

El problema para la computación secuencial es conectar el vacío entre el paralelismo natural que domina todas las aplicaciones y la naturaleza secuencial de la solución software. La solución de este problema parece cada vez más complicado.

El problema para la computación paralela es conectar el vacío entre el software y el hardware paralelo. Valiant ha propuesto el concepto de

Paralelismo de Sincronía Gruesa (BSP) como posible material para la construcción de este puente.

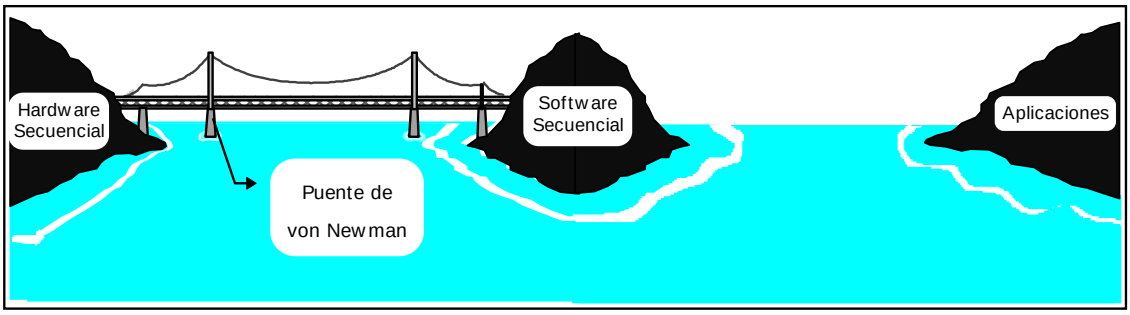


Figura 2

Los principales requerimientos que hacen los usuarios genéricos a la computación paralela, no se ven satisfechos con la construcción de ninguno de los puentes propuestos. La clase de aproximación que se demanda es muy diferente: se pretende la construcción de un puente entre el hardware paralelo y el software secuencial (Figura 3).

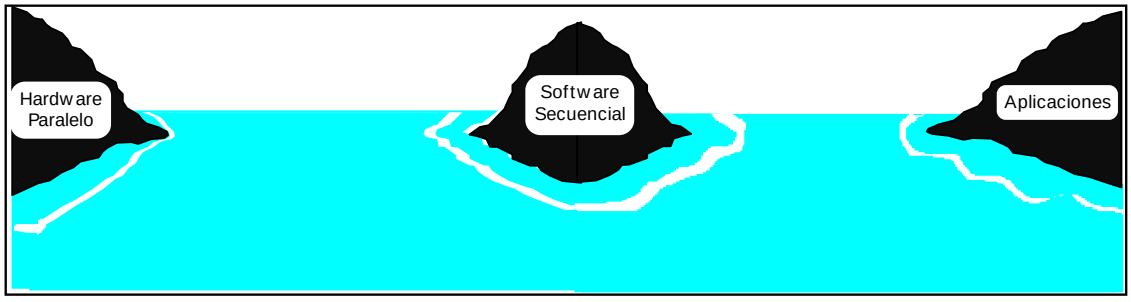


Figura 3

No es necesario pretender que el puente de Valiant establezca una conexión eficiente entre todas las formas de paralelismo software y hardware. En el lado software es necesario un comportamiento determinista

de los procesos (a menos que explícitamente se introduzca no determinismo) y una semántica formal que se corresponda con la intuición. Mientras que desde el punto de vista hardware, es necesario por un lado establecer el equilibrio entre la potencia computacional y el ancho de banda de las comunicaciones y por el otro, el coste en tiempo de la sincronización de procesos, de la latencia de los mensajes y del cambio de contexto entre procesos.

El problema planteado de forma general en los párrafos anteriores se puede resumir, desde el punto de vista de los diseñadores de lenguajes y compiladores en las siguientes cuestiones:

- ¿Cuáles son las necesidades de los programadores?
- ¿Qué características de las máquinas paralelas están más afianzadas y se deben reflejar en los lenguajes de programación?
- ¿Qué características de las máquinas paralelas son irrelevantes para los programas independientes de las máquinas?
- ¿Qué propiedades debería poseer el hardware paralelo para permitir compilaciones eficientes de los programas?

El método tradicional de diseñar algoritmos y formularlos en lenguajes de programación con paralelismo explícito tiene mucho éxito, especialmente si se parte del hecho de que el cuerpo de conocimiento de los algoritmos paralelos es grande y continúa creciendo rápidamente. Las primeras aproximaciones parecen indicar que escribir software paralelo no es significativamente más duro que escribir software secuencial, siempre que se proporcionen los soportes y herramientas adecuados.

Ante la cuestión de cuáles son las demandas generales que se le hacen a los lenguajes de programación y herramientas de soporte paralelas, se pueden extrapolar algunos de los requerimientos del ámbito de la programación secuencial. Podemos enumerar los siguientes:

- Los lenguajes de programación deberían permitir una expresión clara de los algoritmos, facilitar la escritura, lectura, verificación, comprensión, modificación y reutilización de los programas.
- Los programas deberían ser portables a un amplio rango de arquitecturas hardware.
- Los programas deberían ejecutarse (después de la compilación) con eficiencia y con utilización efectiva de los recursos disponibles en hardware.
- Se debería disponer de herramientas de soporte para la depuración de los programas paralelos.

Ante este estado de las cosas, los diseñadores de lenguajes y compiladores se plantean cuestiones concretas como: ¿Cuáles son las propiedades de las arquitecturas paralelas que deberían ser visibles en los programas paralelos?, ¿Debería ser visible el número de procesadores?, ¿Es necesario que el programador conozca la organización de la memoria o la distribución de los datos?, ¿Son necesarios protocolos de comunicación explícitos, o debería el compilador ser capaz de insertarlos en el código generado mediante un análisis de los patrones de datos usados?, ¿Debería ser visible el modo de control?, esto es, ¿deberían los diferentes lenguajes de programación especializarse para máquinas SIMD, MIMD, Flujo de Datos (Dataflow) o sistólicos?

Entre las características de la máquina paralela que tienen que ser visibles al lenguaje de programación se encuentra el número de procesadores, que ha de estar disponible (como una constante o variable), para que los algoritmos se puedan adaptar a él.

No debe forzarse a especificar muchos detalles concernientes a la distribución de los datos en la estructura de la red, puesto que esto tiende a hacer los códigos fuentes dependientes de la máquina. En su lugar, se deben desarrollar herramientas que permitan embeber automáticamente o semiautomáticamente los datos y los patrones de acceso en las estructuras hardware dadas.

Se debería también evitar el paso de mensajes en los lenguajes de programación de alto nivel, puesto que es tedioso, produce errores y las instrucciones del paso de mensajes están íntimamente relacionadas con la manera en la que el problema es aplicado en la arquitectura objeto dada. Entonces, el paso de mensajes tiende a provocar la dependencia de la máquina que es difícil de eliminar y escalar.

El modelo PRAM es claramente la mejor aproximación de la programación paralela estudiada hoy en día. La mayoría de los algoritmos paralelos están formulados para el modelo PRAM. La memoria compartida de este modelo hace los programas mucho más fáciles de comprender que con modelos de memoria distribuida y paso de mensajes. Una idea práctica es que la PRAM sea el modelo de máquina que el programador vea, y que la PRAM sea simulada automáticamente en alguna máquina paralela real, por ejemplo sobre una red de procesadores. Esta es la idea básica de esta tesis. Sin embargo, la proyección del modelo PRAM abstracto sobre una máquina real implica una pérdida de rendimiento importante, puesto que la PRAM ignora completamente la jerarquía de memoria de las máquinas actuales.

El capítulo I introduce el modelo PRAM original propuesto por Fortune y Wyllie. Se hace un estudio de las variaciones que existen sobre este modelo, haciendo especial énfasis en el modelo BSP de Valiant.

En el segundo capítulo se describen varios lenguajes de programación en paralelo que representan otras tantas soluciones a las cuestiones suscitadas en este prólogo. Prestamos especial atención al lenguaje FORK desarrollado en la Universidad de Saarlandes por ser un lenguaje orientado al modelo PRAM con características muy similares al lenguaje que constituye el núcleo de este trabajo: ll.

El lenguaje ll es un lenguaje de propósito general, extensión de un subconjunto de Pascal. ll está orientado a problemas, en el sentido de que el programador puede escoger y mezclar el grado de paralelismo, esto es, el número de procesadores, el modo de control (como una SIMD o una MIMD) tanto como necesite el algoritmo que se está implementando. En ll la red de

interconexión no es visible directamente desde el lenguaje. Se asume la existencia de una memoria compartida por todos los procesadores. No hay paso explícito de mensajes; en su lugar se admiten lecturas y escrituras de la memoria compartida. La primera versión del lenguaje y su compilador se terminaron a finales de 1991 constituyendo el núcleo de la memoria de licenciatura que precedió a este trabajo y que se leyó en 1992. Una versión para redes de transputers fue el resultado de una segunda memoria de licenciatura en 1993 inscrita dentro de esta misma línea de investigación. En el tercer capítulo, se presenta el sistema ll. Se describen las sentencias del lenguaje así como la traducción que se emite para la red de transputers que conforman la fase final (*back-end*) del compilador, se muestran ejemplos prácticos, se introduce un modelo sencillo de complejidad algorítmica que permite predecir el orden del tiempo de computación de un algoritmo escrito en ll usando las versiones 2.0 ó 3.0 y se termina con la discusión de las deficiencias actuales del sistema ll así como presentando las soluciones que se proponen en ll 4.0 para solventar esas deficiencias.

Finalmente debemos indicar que en la redacción de este trabajo ha surgido la necesidad de utilizar numerosos términos del inglés. En todos los casos se propone una posible traducción, sin embargo, cuando no hay una palabra lo suficientemente aceptada se ha optado por el término inglés.

Capítulo I

El Modelo PRAM

1. El Modelo PRAM.

Hoy en día es evidente la necesidad de nuevos lenguajes que permitan una adecuada expresión de las técnicas de resolución de problemas que explotan la potencia del hardware paralelo. En este capítulo se introduce una extensión del modelo de von Newman: El modelo de computación paralela PRAM. Este modelo permite a los diseñadores de algoritmos paralelos tratar la potencia computacional como un recurso ilimitado. El modelo PRAM es simple puesto que ignora la complejidad de las comunicaciones entre procesadores, lo que permite que el diseñador de algoritmos pueda centrarse en el paralelismo inherente a una computación particular. En la primera sección de este capítulo se hace un repaso del modelo clásico de computación secuencial RAM para extenderlo a continuación al modelo PRAM siguiendo la definición original de Fortune y

Wyllie. En el tercer subepígrafe se discuten algunas variantes que se refieren a la forma en la que se resuelven los conflictos de acceso a memoria. La abstracción que supone el modelo PRAM con respecto a las comunicaciones es la razón por la que muchos autores han argumentado la imposibilidad de construir tal modelo. En el cuarto subepígrafe se discute esta cuestión y se aporta una prueba de que la no factibilidad del modelo PRAM, de existir, se debe a factores económicos y no a factores tecnológicos. En las dos secciones siguientes se hace una reseña de las extensiones y restriciones del modelo PRAM que aparecen en la literatura. Por último, se citan los intentos más conocidos de realización de máquinas siguiendo este modelo.

1.1 Un Modelo de Computación Secuencial.

La máquina RAM (**R**andom **A**ccess **M**achine) [Aho74] es un modelo de máquina con un acumulador en el que las instrucciones no se pueden modificar a si mismas.

Una RAM consta de una memoria, una cinta de entrada de sólo lectura, una cinta de salida de sólo escritura y un programa (Figura 1).

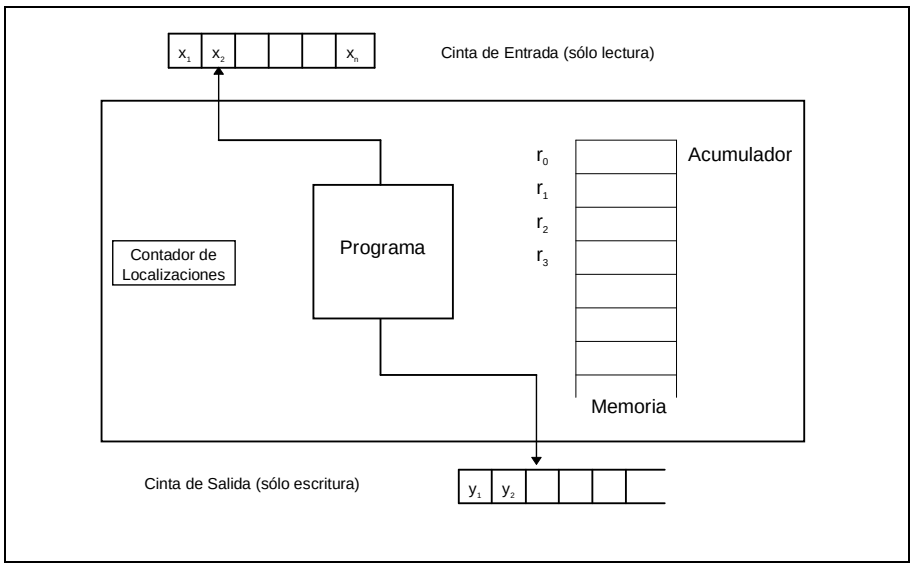


Figura 1. Modelo RAM de Computación Secuencial.

El programa no se almacena en memoria y no puede modificarse. La cinta de entrada contiene una secuencia de enteros, y cada vez que se lee un valor, la cabeza de entrada avanza una posición. De la misma forma avanza la cabeza de escritura. La memoria consta de una secuencia no acotada de registros, denotados por $\{r_0, r_1,...\}$. Cada registro sólo puede almacenar un entero. El registro r_0 es el acumulador donde se realizan todos los cálculos. El juego exacto de instrucciones no es importante puesto que es una semblanza de las instrucciones de los computadores actuales.

La complejidad en tiempo en el ***caso peor*** de un programa RAM es una función *f(n)* que representa el tiempo máximo que consume el programa al ejecutarse sobre todas las entradas de tamaño *n*. La complejidad en tiempo ***esperada*** de un programa RAM es la media de los tiempos de ejecución de entre todas las entradas de tamaño n. Se definen de forma similar la complejidad en espacio en el caso peor y la complejidad en espacio esperada.

El tiempo y el espacio requerido por un programa diseñado para el modelo RAM se puede medir de dos formas:

Mediante el ***criterio de costo uniforme***: en el que cada instrucción RAM requiere una unidad de tiempo para ejecutarse y cada registro requiere una unidad de espacio.

El ***criterio de coste logarítmico***: que tiene en cuenta que una palabra de memoria tiene una capacidad de almacenamiento limitada.

El costo uniforme es el apropiado si los valores manipulados por el programa siempre se pueden almacenar en una palabra de la máquina.

1.2 El Modelo de Computación Paralela PRAM.

El modelo de computación paralela **P**arallel **R**andom **A**ccess **M**achine (PRAM) propuesto por Fortune y Wyllie [For78] en su forma original consta de: un conjunto no acotado de procesadores $\{P_0, P_1,...\}$, una memoria global no acotada, un conjunto de registros de entrada y un programa finito.

Cada procesador tiene un acumulador (A), una memoria local no acotada (R_i), un contador de programa (PC) y una bandera (F) que indica si el procesador está trabajando o no. Todas las localizaciones de memoria y acumuladores permiten el almacenamiento de enteros no negativos arbitrarios. El programa consiste en una secuencia de instrucciones del juego de la Tabla I, que pueden estar o no etiquetadas.

Instrucción	Significado
LOAD operando STORE operando ADD operando SUB operando	Realiza la operación indicada utilizando el acumulador del procesador que la ejecuta.
JUMP etiqueta JZERO etiqueta	Cambia el contador de programa a etiqueta
READ operando	Véase el texto
FORK etiqueta	Véase el texto
HALT	Véase el texto

Tabla I. Juego de Instrucciones del Modelo PRAM

Cada operando puede ser: Un literal, una dirección o un direccionamiento indirecto. Cada procesador puede acceder a la memoria global o a su memoria local, pero no a la memoria local de cualquier otro procesador. Se puede realizar direccionamiento indirecto sobre una posición de memoria para acceder a otra.

Inicialmente la entrada para la PRAM se coloca en los registros de entrada, un bit por registro, se limpia toda la memoria, la longitud de la entrada se pone en el acumulador de P_0 y se inicializa P_0. En cada paso de la computación todos los procesadores activos ejecutan simultáneamente la instrucción indicada por su contador de programa en una unidad de tiempo, e incrementan su contador en una unidad a menos que la instrucción provoque un salto.

Una instrucción de lectura *READ operando* utiliza el valor del operando para especificar uno de los registros de entrada, el contenido del registro seleccionado se coloca en el acumulador.

Una instrucción *FORK etiqueta* ejecutada por el procesador P_i selecciona el primer procesador inactivo P_j, limpia la memoria local de P_j, copia el acumulador de P_i en P_j e inicializa el contador de programa de P_j a *etiqueta*.

Una instrucción *HALT* provoca que un procesador detenga su ejecución.

Se permiten lecturas simultáneas de una posición de la Memoria Global, pero si dos procesadores intentan escribir en la misma posición de memoria simultáneamente la PRAM se detiene inmediatamente y rechaza tal operación como lícita.

Varios procesadores pueden leer una posición de memoria mientras un procesador escribe en ella, todas las lecturas se realizan antes de que el valor de la posición se cambie.

La ejecución continúa hasta que P_0 ejecuta un *HALT* (o dos procesadores intentan escribir en la misma posición simultáneamente).

Nótese que, incluso si el programa es el mismo, en un instante dado procesadores diferentes pueden ejecutar instrucciones diferentes. De hecho se ha visto que cada procesador se puede activar comenzando su ejecución desde cualquier instrucción del programa. Además, se debe considerar la presencia de instrucciones de ramificación condicional, para las que el resultado es en general diferente en cada procesador. Esto justifica la existencia de un registro contador de programa en cada procesador. A pesar de esto el modelo PRAM es síncrono, y todos los procesadores inician la ejecución de sus respectivas instrucciones en el mismo instante.

1.3 Variaciones sobre el Modelo Original.

En la sección anterior se describía la arquitectura del modelo PRAM original. Sin embargo en la literatura sobre algoritmos paralelos los programas PRAM no se detallan a ese nivel. En la mayoría de los casos se hace una descripción a un nivel más alto y más legible. Generalmente se hacen consideraciones en la redefinición del modelo en orden de hacerlo más manipulable y más fácil desde el punto de vista del diseñador de algoritmos.

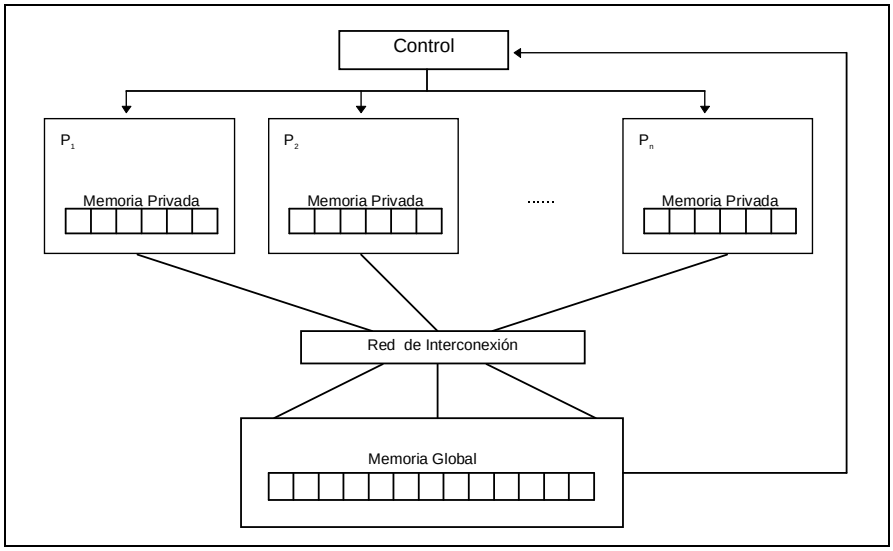


Figura 2. El modelo PRAM de Computación Paralela.

Proporcionar los valores iniciales a una computación con $P(n)=n^{O(1)}$ procesadores, donde n es el tamaño de la entrada, requiere tiempo $O(\log n)$. Es imposible obtener tiempos paralelos sublogarítmicos para estas computaciones. Sin embargo, si la computación en sí misma (asumiendo que los procesadores están activos inicialmente) tiene un costo $\Omega(\log n)$ es entonces evidente que la fase de inicialización no influye en el orden de magnitud del costo del tiempo. Una vez clarificados estos puntos, es absolutamente innecesario incluir una fase de inicialización en cada

programa PRAM. En consecuencia, se asume que todos los P(n) procesadores están inicialmente activos. Como consecuencia, la instrucción FORK se excluye del juego de instrucciones. Tampoco es necesaria la presencia de un registro indicador del estado.

Un procesador genérico P_i recibe un número identificador j_i (único), cuando se inicializa, que no necesariamente coincide con i. Es evidente que no se pierde generalidad si siempre se emplea $j_i = i$. Por lo que es común asumir que cada procesador puede leer su propio número de identificación desde un registro especial de sólo lectura.

Se puede asumir también que el tamaño de la entrada, n, es conocido por todos los procesadores que participan en la computación.

Existen dos formas básicas de funcionamiento en el modelo de memoria compartida. En la primera forma, denominada ***síncrona***: todos los procesadores operan síncronamente bajo el control de un reloj común. Esto es, todos los procesadores ejecutan el mismo programa de tal manera que en cada unidad de tiempo todos los procesadores activos están ejecutando la misma instrucción pero en general sobre datos diferentes; este modelo de memoria compartida es del tipo Una Instrucción-Múltiples Datos (**SIMD**).

En la segunda forma, denominada ***asíncrona***: cada procesador opera bajo un reloj separado. En el modo de trabajo asíncrono, es responsabilidad del programador determinar y especificar los puntos de sincronización necesarios. Puesto que cada procesador puede ejecutar su propio programa local, este modelo de memoria compartida es del tipo Múltiples Instrucciones-Múltiples Datos (**MIMD**). Esto es, cada procesador puede ejecutar una instrucción diferente u operar sobre datos diferentes de los restantes procesadores en cualquier unidad de tiempo.

Los algoritmos desarrollados para el modelo PRAM en general son del tipo SIMD, sin embargo, como el modelo establece, se pueden cargar programas diferentes en la memoria local de los procesadores, siempre que los procesadores puedan operar síncronamete, entonces, instrucciones

diferentes se pueden ejecutar dentro de la unidad de tiempo asignada a un paso de computación.

El aspecto más importante concierne a la sincronía en la actividad de los procesadores. Incluso cuando se tratan ejemplos simples, tales como el cálculo de $\lceil \log n \rceil$ o la suma de n números, no es fácil asegurar que todos los procesadores interactuaran de la forma correcta y sobre todo, no hay garantías de que los accesos a la memoria global no violen alguna restricción (por ejemplo, escrituras simultáneas). Realmente, aunque la PRAM es síncrona, las características de programación son asíncronas. Por esta razón, el modelo PRAM original ha evolucionado a modelos donde todos los procesadores son forzados a ejecutar la misma instrucción. En otras palabras, existe un único contador de programa global. Los programas se especifican usando lenguajes de programación de alto nivel en los que la sincronización se puede forzar de forma implícita. Por ejemplo, la instrucción:

$$\text{if name} = i \text{ then } m_i \leftarrow m_i + 1$$

ejecutada por todos los procesadores, es tal que sólo el procesador P_i asigna $m_i \leftarrow m_i + 1$, mientras los restantes procesadores ejecutan un bucle de sincronización. Esto también se aplica a las sentencias iterativas.

Otro aspecto de arquitectura considerado en la literatura que proporciona una serie de variaciones sobre el modelo PRAM original es la forma de realizar los accesos a la memoria global.

En el modelo PRAM original más de un procesador puede leer simultáneamente el contenido del mismo registro de memoria global, mientras que todo intento de escribir simultáneamente se detiene inmediatamente mediante la ejecución de una instrucción *HALT*. Una PRAM que siga este criterio en los accesos a la memoria global se denomina una ***CREW PRAM*** (**C**oncurrent **R**ead **E**xclusive **W**rite).

También son posibles las otras dos soluciones. En el modelo ***EREW PRAM*** (**E**xclusive **R**ead **E**xclusive **W**rite) también se detiene la ejecución del

programa cuando se intenta una lectura al mismo registro. Al contrario que en las ***CRCW PRAM*** (**C**oncurrent **R**ead **C**oncurrent **W**rite) en la que se admiten también escrituras concurrentes en la misma posición de memoria. En este caso, para asegurar el determinismo de la computación, se requiere un criterio que establezca un contenido único para la celda de memoria global en la que tendrá lugar la operación de escritura. En la literatura aparecen las siguientes versiones del CRCW PRAM:

Weak. Más de un procesador puede escribir simultáneamente en la misma posición, sólo si el valor que escribe es un 0.

Common mode. Más de un procesador puede escribir simultáneamente en la misma localización, sólo si el valor a escribir es el mismo para todos.

Arbitrary winner. El valor que se escribe es arbitrario, y no se puede determinar a priori cuál de los valores presentados por los procesadores será.

Priority mode. El valor que se escribe es el presentado por el procesador con el mayor identificador (o de forma equivalente el menor identificador).

Strong. El valor que se escribe es el mayor (o de forma equivalente el menor).

También se ha mencionado que varios procesadores pueden leer una localización de memoria mientras un procesador escribe en ella y en el modelo PRAM original se asume que todas las lecturas se realizan antes de que el valor de la posición se cambie. La alternativa sería el considerar que la escrituras tienen lugar antes que las lecturas. Se tendrán entonces nuevas variantes de los modelos ya mencionados añadiendo los prefijos *FR* (**F**irst **R**ead, que coincide con el modelo original) y *FW* (**F**irst **W**rite).

1.4 Viabilidad del Modelo PRAM.

El modelo de computación paralela PRAM abandona todas las restricciones hardware que una arquitectura altamente específica pudiera imponer. A este respecto el modelo da libertad en la presentación de algoritmos y no admite las limitaciones sobre el paralelismo que podría imponer un hardware específico. Incluso en su manifestación más simple la EREW PRAM es bastante potente. Esto implica que en la realización de una PRAM existirían todos los enlaces posibles entre procesadores y localizaciones de memoria global. Hay quien opina que este modelo no es realista, y que no es posible construir un ordenador paralelo basado en él con la tecnología actual [Gib88, Akl89a]. Akl [Akl89a] lo argumenta de la siguiente manera: cuando un procesador necesita acceder a un dato en memoria, es necesario que se cree un camino físico desde el procesador hasta la posición de memoria, se necesitarán puertas lógicas que decodifiquen la dirección. Si la memoria tiene *M* posiciones, el coste de la circuitería de decodificación se expresaría como *f(M)* siendo *f* alguna función de coste. Si *N* procesadores comparten esa memoria, el coste sube a $N \times f(M)$ para *N* y *M* grandes ese coste puede ser excesivo.

Se proponen algunas formas de mitigar el problema, y todas ellas implican simplificar el modelo PRAM. Sin embargo, es conocido que cualquier algoritmo para el modelo PRAM puede implementarse para modelos más simples a costa de aumentar la memoria y el tiempo de ejecución. Por supuesto, cualquier algoritmo para modelos que son simplificaciones del PRAM, correrá sobre una máquina PRAM sin coste adicional.

Una forma de reducir el coste de la circuitería de decodificación de direcciones consiste en dividir la memoria compartida en *R* bloques de, por ejemplo, *M/R* posiciones cada uno. Habrá $N = R$ líneas de doble dirección que permitan a cualquier procesador acceder a cualquier bloque en cualquier momento. Sin embargo, solamente un procesador en cada

momento podrá escribir o leer en un bloque. Esta estructura se presenta en la Figura 3 para *N=5* y *R=3*. Los puntos en las intersecciones entre líneas horizontales y verticales representan pequeños conmutadores (relativamente baratos). Cuando el procesador *i*-ésimo quiere acceder al *j*-ésimo bloque, manda una petición por la línea horizontal *i* hasta el conmutador *j*, quien lo reencamina por la línea vertical hacia el bloque de memoria. Cada bloque de memoria tiene un circuito decodificador para determinar cual de las *M/R* posiciones de memoria es la requerida. Así el coste total de la circuitería de decodificación es $R \times f(M/R)$, a lo que hay que sumar el coste de los $N \times R$ conmutadores.

Creemos que la no factibilidad del modelo PRAM es debida más a limitaciones de costo que a limitaciones tecnológicas. La debilidad del razonamiento dado por Gibbons y Rytter se basa en el supuesto de que las celdas de memoria global no son replicadas.

En el esquema de la Figura 4 hacemos una propuesta para la realización de la memoria global compartida. Cada dirección de memoria *d* se replica en cada procesador así que la dirección de memoria *d* está representada por el conjunto de celdas en la columna etiquetada d. Cuando un procesador P_i realiza un acceso para lectura simplemente lee de su réplica de la dirección correspondiente. Cuando un valor es almacenado por un procesador P_i en su réplica de la dirección compartida *d*, la celda receptora trasmite el nuevo valor a las otras celdas replicadas utilizando el enlace de emisión (*broadcast*) *d*. En general el tiempo invertido en la emisión (*broadcasting*) será una función *BT(p)* del número *p* de procesadores. Por ejemplo, cuando las celdas replicadas pertenecientes a la misma dirección son conectadas de acuerdo a una topología árbol, el tiempo de emisión será $BT(p) = \log p$. Si se asume una *BT(p)* constante, el modelo que se acaba de describir tiene el mismo poder computacional que el modelo PRAM clásico.

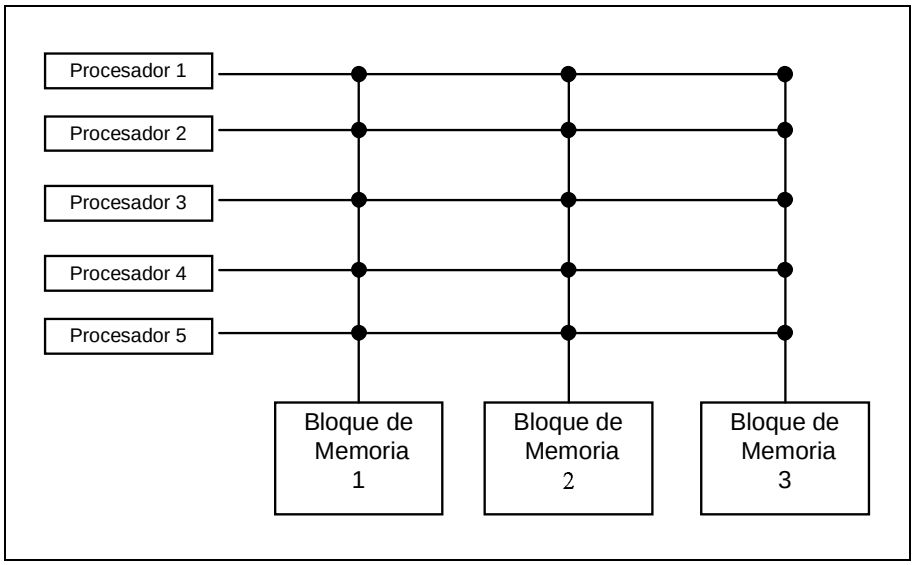


Figura 3. División de la Memoria Compartida en Bloques.

Obviamente el coste del hardware para construir esta realización para un número *p* de procesadores y una memoria de tamaño *M*, es proporcional a *p x M.* Este resultado concuerda con la predicción que se sigue del razonamiento expuesto en párrafos anteriores debido a Akl. En consecuencia nuestra propuesta no es factible desde un punto de vista económico para valores de *p* y *M* grandes. Dicho costo se puede reducir si se utilizan bloques de memoria en lugar de celdas de manera que se disminuya el número de enlaces de emisión. Las lecturas del mismo bloque no provocan secuencialización, mientras que las escrituras sí.

Seguramente las nuevas tecnologías cambiarán nuestra percepción de lo que es físicamente realizable y lo que no. Sin embargo, debido a su simplicidad y universalidad, nosotros creemos que el modelo PRAM sobrevivirá como un modelo teórico de la computación paralela y como un punto de partida para una metodología de programación.

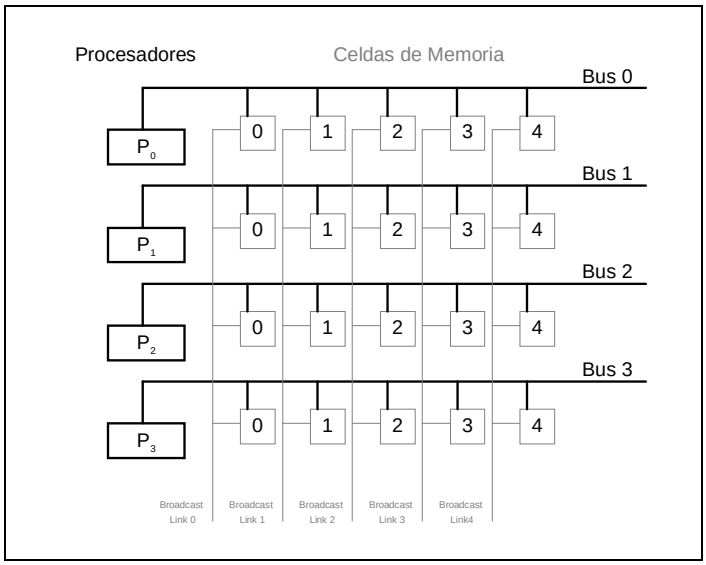


Figura 4. Esquema de la Memoria Compartida.

En los siguientes epígrafes se aborda la tarea de describir de manera somera algunos de los modelos de memoria compartida que aparecen en la literatura. Estas variantes del modelo PRAM se agrupan en dos grandes categorías: aquellos que introducen restricciones en el modelo para hacerlo factible y los que lo expanden con nuevas propiedades buscando facilitar la expresión de algoritmos paralelos.

2. Restricciones del Modelo PRAM.

A continuación se examinan ciertos modelos inspirados en el modelo básico PRAM , pero que son más restrictivos, cada uno en algún sentido particular. Los investigadores que proponen estos modelos argumentan como motivación que la PRAM básica es demasiado potente, y que un modelo más restrictivo podría ser más realista desde el punto de vista tecnológico.

2.1 Accesos por Prioridad.

En el modelo CROW (Concurrent Read, Owner Write) cada celda de memoria compartida pertenece a un único procesador que se denomina su propietario. Un procesador solamente puede escribir en las celdas que le pertenecen. Todo procesador puede calcular fácilmente las celdas que le pertenecen. Este modelo ha sido generalizado haciendo variar la función de propiedad con el tiempo. En la mayoría de los casos es sencillo convertir un algoritmo CREW PRAM en un algoritmo CROW PRAM. De acuerdo con [Dym87] el modelo CROW se adapta mejor a ciertos ordenadores existentes.

La idea de propiedad se ha extendido para las lecturas en [Ros90]. Se obtienen nuevas variantes de acceso al aplicar los modos de acceso propietario, exclusivo y concurrente . En concreto, en el modelo OROW PRAM (Owner Read, Owner Write) podría aparecer una contradicción con el concepto de memoria compartida. La contradicción se elimina si se permite variar la función de propiedad de una etapa a otra. Los autores del mencionado trabajo afirman que el modelo propietario es más realista que el exclusivo y el concurrente. Esta opinión se basa en la hipótesis de que aparentemente toda implementación de una memoria compartida estaría basada en una memoria distribuida.

2.2 RW-PRAM: Palabras de tamaño limitado.

La mayor parte de los estudios relativos al modelo PRAM suponen que cada dirección de memoria compartida tiene espacio suficiente para almacenar un entero arbitrario. La RW (Restricted Wordsize) PRAM introducida en [Bel91] limita el número de bits que se pueden almacenar en cada celda de memoria compartida. En dicho trabajo se demuestra que las concepciones de la PRAM que almacenan la misma cantidad de información pero que tienen palabras de tamaños diferentes no son comparables. En efecto, dado un modelo, una reducción en el tamaño de una palabra no se puede compensar siempre con el

incremento correspondiente en el número de palabras o en el número de procesadores, o con permitir una regla más potente en la resolución de los conflictos de escritura.

2.3 El Modelo BSP.

El modelo BSP y los modelos relacionados tienen como objetivo proporcionar un punto de vista sencillo de una máquina paralela de tal manera que el programador sea capaz de diseñar y analizar algoritmos cuyo rendimiento sea predecible en las máquinas reales. Sin embargo, el modo en que se visualiza la computación, así como el modo en se lleva a cabo la ejecución del algoritmo puede diferir de un modelo a otro. Al programador se le presenta una jerarquía de memoria de dos niveles, cada procesador tiene su propia memoria local o privada y puede acceder a una memoria común compartida. Desde un punto de vista lógico la memoria compartida es un único espacio de direcciones aunque físicamente puede estar implementada a través de un rango de memorias distribuidas.

El modelo BSP (***B****ulk* ***S****ynchronous* ***P****arallelism*) de computación paralela [Val90a] se define como la combinación de tres atributos:

1. Un conjunto de *Componentes*, en el que cada una tiene capacidad de realizar funciones de proceso y/o accesos a memoria.

2. Un *Mecanismo de Comunicaciones* (*router*) que entrega mensajes punto a punto entre pares de componentes; y

3. Recursos para *sincronizar* todos o un subconjunto de componentes a intervalos regulares de *L* unidades de tiempo, donde *L* es el *Parámetro de Periodicidad.*

Una computación consiste en una secuencia de *superpasos*. En cada superpaso, cada componente tiene asignada una tarea que consiste en alguna combinación de computaciones locales, transmisión de mensajes y llegada (implícita) de mensajes desde otras componentes.

Después de cada período de *L* unidades, una comprobación global determina si todas las componentes han completado el superpaso. Si es el caso, la máquina procede con el siguiente superpaso. En otro caso, el siguiente período de *L* unidades se asigna al superpaso sin acabar. El mecanismo de sincronización se puede "*desactivar*" sobre cualquier subconjunto de componentes; los procesos secuenciales que son independientes de los resultados de procesos en otras componentes no deberían ser innecesariamente retrasados. Cuando la sincronización se desactiva en un procesador, éste procederá sin tener que esperar por la finalización de los procesos o las comunicaciones en otras componentes. Así mismo, las computaciones locales al procesador no deberán retrasar las computaciones en ninguna otra componente. Por otra parte, incluso cuando este mecanismo de sincronización por barrera está desactivado, un procesador puede todavía enviar y recibir mensajes. Este envío y recepción de mensajes proporciona un método alternativo de sincronización. Si se esperan garantías de un mejor rendimiento al utilizar este mecanismo alternativo de sincronización, será necesario hacer suposiciones acerca del mecanismo de comunicaciones, por ejemplo, se puede asumir que cada mensaje es entregado dentro de una cierta cantidad de tiempo desde el comienzo de su envío.

Al justificar el modelo BSP se utiliza el mecanismo de sincronización por barrera y no se hacen suposiciones acerca de los tiempos relativos de entrega dentro de un superpaso. En las simulaciones, las operaciones locales se realizan únicamente con datos disponibles localmente antes del arranque del superpaso actual. El valor de periodicidad *L* puede ser controlado por el programa, incluso en el momento de ejecución. La elección de este valor está condicionada de maneras opuestas por consideraciones de hardware y de software. Claramente, el hardware establece cotas inferiores de lo pequeño que pueda ser *L*. El software, por otro lado establece cotas superiores de *L*, puesto que cuanto mayor es, mayor será la granularidad de paralelismo que deba tener el programa. Esto es porque, para alcanzar un uso óptimo de los procesadores, en cada

superpaso cada procesador tiene que tener una tarea de aproximadamente *L* pasos en la que pueda proceder sin tener que esperar por los otros procesadores.

Cuando se analiza el rendimiento de un computador BSP, se asume que una componente de proceso puede computar una operación sobre datos disponibles en su memoria local en una unidad de tiempo. La tarea básica del mecanismo de comunicaciones es realizar *h-relaciones* o, en otras palabras, superpasos en los cuales cada componente envía o recibe a lo sumo *h* mensajes. Se presisa una carga de *gh+s* unidades de tiempo para realizar tal h-relación, donde *g* define el "*throughput*" básico de la red cuando está en uso continuo y *s* es la latencia (o coste de "*startup"*). Puesto que únicamente interesan las simulaciones óptimas se asume que *h* es suficientemente grande para que *gh* sea al menos comparable con *s*. Si $gh \geq s$, por ejemplo, y se toma $g = 2g$, entonces se pueden cargar *gh* unidades de tiempo por una h-relación y esto será una sobreestimación (por un factor de a lo más dos). Sea *g* tal que las h-relaciones pueden realizarse en tiempo *gh* para *h* mayor que algún h_0 . Este *g* puede verse como la razón entre el número de operaciones locales realizadas por unidad de tiempo por todos los procesadores y el número total de palabras entregadas por segundo por el mecanismo de comunicaciones. Nótese que si $L \geq gh_0$ entonces cualquier h-relación con $h < h_0$ tendrá un costo como el de una h_0-relación.

Se puede considerar que incluso fijando la tecnología el valor del parámetro *g* se mantiene dentro de ciertos límites. El valor *g* se puede mantener bajo utilizando más segmentación (*pipelining*) o proporcionando canales de comunicación más anchos. Mantener *g* bajo o fijo cuando el tamaño *p* de la máquina crece incurre, por supuesto, en costes extra. En particular cuando la máquina escala las necesidades de inversión para las comunicaciones crecen más rápidas que las de la computación. La idea de Valiant es que, si estos costes se pagan se pueden alcanzar máquinas con un nuevo nivel de programabilidad y eficiencia.

De acuerdo con Valiant, el método mas prometedor para distribuir los accesos a memoria de manera equilibrada en programas arbitrarios es el *hashing*. La idea que lo motiva es que si las palabras de memoria son distribuidas entre las unidades de memoria aleatoriamente, independientemente del programa, entonces los accesos a las distintas unidades deberían tener la misma frecuencia. Melhorn y Vishkin [Meh84] han propuesto funciones *hash* para este contexto paralelo. Estos autores han sugerido una clase de funciones con algunas propiedades deseables: la clase de los polinomios de grado O(log p) en la aritmética módulo *m*, donde *p* es el número de procesadores y *m* es el número de palabras en el espacio de memoria.

Para que el *hashing* tenga éxito en algoritmos paralelos que se ejecutan con eficiencia óptima es necesaria cierta holgura paralela, y si *g* puede verse como una constante es suficiente una cantidad moderada. Para ver esta necesidad se debe tener en cuenta que si se hacen únicamente *p* accesos en un superpaso a *p* componentes aleatorias, existe una alta probabilidad de que una componente obtenga alrededor de $\log p / \log \log p$ accesos y algunas ninguno. Por tanto, la máquina tendrá que dedicar no menos de $\log p / \log \log p$ unidades de tiempo en lugar de tiempo constante, que sería lo necesario para un *throughput* óptimo. El lado positivo es que si se hacen unos pocos más, por ejemplo, $p \log p$ accesos aleatorios en un superpaso, existe una gran probabilidad de que cada componente no obtenga más de $3 \log p$. Por tanto, estos accesos pueden ser implementados por el mecanismo de comunicaciones en la cota óptima *O(log p).* Este fenómeno puede explotarse de la siguiente forma:

Supóngase que cada una de las *p* componentes de una máquina BSP consta de una memoria y un procesador. Se simula un programa paralelo con $v \geq p \log p$ procesadores virtuales asignando a cada procesador físico la simulación de $v/p \geq \log p$ procesadores virtuales. En un superpaso, la máquina BSP simula un paso de cada procesador virtual. Entonces las *v* peticiones de memoria se extenderán paritariamente,

alrededor de *v/p* por procesador, por lo tanto la máquina será capaz de ejecutar este superpaso en tiempo óptimo O(v/p) casi seguro. Este análisis supone que las *v* peticiones son a localizaciones de memoria distintas. Para mantener las constantes de proporcionalidad bajas la máquina tendrá que ser eficiente tanto en hashing como en conmutación de contexto (*context switching*).

En conclusión, para explotar el hashing de forma eficiente, la periodicidad *L* debe ser al menos logarítmica, y si es logarítmica, se puede alcanzar la optimalidad. Al hacer esta afirmación se está considerando como tiempo constante la sobrecarga de evaluar las funciones hash incluso en tiempo de ejecución. Para justificar esta decisión se puede argumentar que la evaluación de la función hash se puede hacer localmente y por tanto rápidamente. Se requieren O(log log p) pasos para evaluar polinomios de grado log p y esto puede considerarse como tiempo constante. La frecuencia de evaluación de las direcciones más utilizadas se puede reducir en la práctica almacenando las direcciones en tablas.

En general, cualquier máquina real BSP impondrá una cota superior en *p*, el número de procesadores, así como una cota inferior en el valor de *g* que puede alcanzarse. También, cualquiera que sea el valor de *g*, se puede deducir una cota inferior de *L*. Se puede imaginar software BSP escrito de tal modo que el código compilado no depende solamente del tamaño del problema, *n*, sino que también dependa de los parámetros *p*, *g* y *L* lo que lo haría más portable.

Cuanto menores sean los valores de *L* y *g*, menores serán los costos de las comunicaciones. El caso extremo es el modelo PRAM donde se asume que las operaciones globales consumen el mismo tiempo que las operaciones locales. Sin embargo, en las máquinas paralelas actuales, estos valores son considerablemente grandes y dependen de los patrones de acceso a memoria global.

2.4 Los Modelos SPRAM y S*PRAM .

El modelo BSP se puede implementar de manera óptima en un modelo sencillo de computación sugerido por las posibilidades de la tecnología óptica [Val90b]. En este modelo en cada paso cada una de las *p* componentes puede transmitir un mensaje dirigiendo un rayo de luz a una componente elegida. Si una componente recibe exactamente un mensaje lo confirma y la transmisión se considera exitosa. Por otro lado, si más de un rayo se dirige a un nodo, ninguno de los mensajes se recibe con éxito en ese nodo y la ausencia de confirmación informa al emisor del fallo. Usando una periodicidad $L \geq \log p$, suponiendo que cada procesador envía a lo sumo log p mensajes y cada uno recibe aproximadamente el mismo número y que no hay otro patrón de detección a la petición global de comunicación; existe un algoritmo aleatorio de Anderson y Miller [And88] que puede adaptarse para realizar esta comunicación sobre este modelo óptico en tiempo O(log p). Así pues, si ese tiempo corresponde a *g* unidades de tiempo, este modelo puede simular una máquina BSP con no menos de $p \log p$ procesadores en tiempo óptimo.

En los modelos que se citan a continuación se utiliza una memoria compartida no uniforme dividida en un número de módulos igual al número de procesadores. El modelo SPRAM (***S**eclusive PRAM*) , también denominado DCM (*Direct Connection Machine*) [Kru90] aplica las reglas de una EREW PRAM a cada módulo (y no solamente a cada celda). De esta manera, un procesador en una etapa puede acceder como máximo a un módulo (y por supuesto a una celda) de memoria. El modelo S*PRAM, basado en la tecnología óptica expuesta, aplica las reglas de la CRCW PRAM a cada módulo. En el caso de escrituras concurrentes, sólo un procesador tiene éxito y cada procesador puede determinar en tiempo constante si su ensayo ha sido rechazado.

2.5 El Modelo XPRAM.

Una XPRAM [Nas93a] con *p* procesadores ejecuta las operaciones en superpasos. En cada superpaso cada procesador ejecuta un cierto número de instrucciones locales y envía o recibe algunos mensajes que implementan operaciones globales de lectura y escritura. Considérese que en una ejecución de un superpaso el procesador *i* realiza a_i operaciones locales, envía b_i mensajes y recibe c_i mensajes desde otros procesadores. Sea $r_i = g^{-1}a_i + b_i + c_i$. Entonces si

$$t = max\{r_i / i \in <p>\},$$

se considera que este superpaso consume $\lceil t/L \rceil L$ operaciones estándar globales, o que su tiempo de ejecución es $\lceil t/L \rceil Lg$. Todos los procesadores saben si se ha completado o no un superpaso al final de cada L operaciones globales. Dentro de cada período, sin embargo, no existe sincronización entre los procesadores. La XPRAM difiere del modelo PRAM en que los procesadores operan asíncronamente y soportan múltiples procesos. Esto lleva a una visión de los datos compartidos como “débilmente coherentes” de manera que la coherencia de los datos sólo se garantiza en un punto de sincronización. La sincronización entre los procesos toma la forma de una sincronización gruesa, garantizando un punto en el algoritmo del cual ningún proceso puede pasar hasta que los otros lo hayan alcanzado. Esto refleja directamente al modelo BSP. La sincronización también utiliza variables futuras, bloqueando un proceso lector hasta que el valor futuro esta escrito.

2.6 El Modelo LogP.

En contraste con la naturaleza de grano grueso síncrono de los algoritmos BSP, la ejecución sobre el modelo LogP [Cul93] puede visualizarse como la ejecución de un cierto número de procesos asíncronos entre sí. La comunicación y la sincronización se realizan a través del paso de mensajes. Un mensaje enviado a un procesador puede ser utilizado tan

pronto como llega, en vez de tener una operación de barrera como en el modelo BSP.

El modelo LogP define cuatro parámetros principales: *P*, *o*, *L* y *g*. *P* representa el número de procesadores. El parámetro *o* está definido como la sobrecarga asociada con la transmisión o recepción de un mensaje. Los parámetros *L* y *g*, aunque utilizan el mismo nombre que en el modelo BSP, tienen significados diferentes. El parámetro *L* establece una cota superior de la latencia en enviar un mensaje pequeño, mientras que *g* se define como el período mínimo (*gap*) entre transmisiones o recepciones consecutivas. El recíproco de *g* es el ancho de banda por procesador. El parámetro *g* es similar al del modelo BSP, en el sentido de que define una medida de la eficiencia en la entrega de mensajes. Sin embargo, no existe sincronización implícita en el modelo LogP, la noción de superpasos realizando h-relaciones no se aplica a este modelo. El modelo supone que la red tiene capacidad finita, esto es, cada procesador no puede tener más de *L/g* mensajes pendientes en la red de cada vez. Los procesadores que intentan exceder este límite se paran hasta que pueda iniciarse la transferencia del mensaje. Esto contrasta con el modelo BSP en el cual cualquier suceso de comunicación equilibrada puede realizarse en tiempo *gh*.

El rendimiento de un algoritmo LogP se puede cuantificar sumando todos sus costos de computación y de comunicación. Los costos de comunicaciones se calculan en términos de la ejecución de mensaje primitivos. Por ejemplo, el coste de leer en una localización remota es *2L+4o*. Se requieren dos mensajes de transmisión, uno para solicitar y otro para enviar el dato. En cada transmisión el procesador implicado en la operación gasta "*o*" unidades de tiempo interactuando con la red y el mensaje consume *L* unidades de tiempo para llegar a su destino. El coste de una operación de escritura es el mismo, puesto que en este caso la respuesta es la confirmación (*acknowledge*) requerida para obtener la consistencia secuencial. Este análisis asume que el dato cabe en un único

bloque de transmisión. Cuando se trata con un bloque de *n* datos básicos, el coste es *2L+4o+(n-1)g*, supuesto que *o* < *g*. Esto es debido a que después de la primera transmisión, las subsiguientes transmisiones tienen que esperar *g* unidades de tiempo. El modelo LogP impone la utilización de comunicaciones equilibradas evitando que el procesador se inunde con mensajes de entrada, puesto que en esta situación todos los procesadores menos los *L/g* que envían, se pararían.

2.7 El Modelo WPRAM.

El modelo WPRAM considera los modelos BSP y LogP como modelos de arquitectura. En este sentido, la siguiente descripción se restringe a este nivel de abstracción. El modelo WPRAM intenta extender el modelo BSP flexibilizándolo. Una diferencia importante es que la sincronización por barrera no se soporta directamente, en su lugar se puede utilizar el paso de mensajes para implementar cualquier operación de sincronización. Esto hace que el modelo WPRAM esté mas próximo al modelo LogP. Sin embargo, mientras que los modelos BSP y LogP se pueden aplicar al diseño de una amplia clase de máquinas, el modelo WPRAM se ha pensado para una clase de máquinas escalables de memoria distribuida. Esto significa que la latencia de la red debería incrementarse en una tasa logarítmica con respecto al número de procesadores, esto es, $D = O(\log(p))$, y que cada procesador debería ser capaz de enviar mensajes a la red con una frecuencia constante, es decir, $g = O(1)$. Los parámetros *D* y *g* son similares a los parámetros *L* y *g* definidos para el modelo LogP. Sin embargo, el parámetro D, en lugar de considerarse una cota superior dada por una constante, representa un retraso medio que se incrementa logarítmicamente con el número de procesadores. Los efectos de contención provocados por las conmutaciones y por el uso de datos compartidos en el procesador de destino quedan modelizados con esta definición de D. El parámetro *g* es el mismo que en el modelo LogP, aunque no se impone límite al número total de mensajes pendientes que un procesador puede tener en la red. El análisis de algoritmos en el modelo WPRAM es más simple que en el modelo LogP puesto que no hay que preocuparse por la

capacidad límite de la red ya que ésta es capaz de manipular una frecuencia máxima de accesos por procesador. Además de los parámetros globales *L* y *g*, el modelo WPRAM define un cierto número de parámetros de la máquina.

En la Universidad de Leeds [Nas95] se ha desarrollado un simulador WPRAM. El simulador está basado en la interacción de procesos que son utilizados para representar a los nodos de la máquina objeto y a los procesos definidos por el usuario. Los algoritmos se implementan utilizando una interface de programación y a continuación se pueden ejecutar en el simulador, de manera que la secuencia de operaciones generada por el programa conduce al simulador (*execution-driven discrete-event simulation*). La arquitectura WPRAM para el simulador es una máquina de memoria distribuida que soporta acceso global uniforme mediante el uso de aleatorización de los datos (Figura 5). El uso de aleatorización, cuando los datos están distribuidos entre las memorias locales evita la necesidad de un encaminamiento aleatorio. El simulador incluye un modelo de rendimiento detallado para el T9000 y el encaminador (router) C104. Las operaciones locales modeladas por el simulador incluyen cálculo aritmético, cambios de contexto, manipulación de mensajes y gestión de procesos locales. Los mensajes que entran a la red se suponen descompuestos para garantizar que ningún mensaje ocupa el conmutador por largos periodos de tiempo, mientras que los datos globales se suponen aleatorizados basados en la unidad de *cache line*. Los costos de las operaciones globales están basados en los parámetros de alto nivel *g* y *D*. El valor de *g* se obtuvo en un proyecto Esprit PUMA y *D* se derivó de una simulación del C104 llevada a cabo en Inmos. El simulador está escrito en C y proporciona una rica interface de programación para ejecutar algoritmos escritos en C. La interface de programación consiste en un conjunto de librerías que soportan gestión de procesos, acceso a datos compartidos y sincronización de procesos.

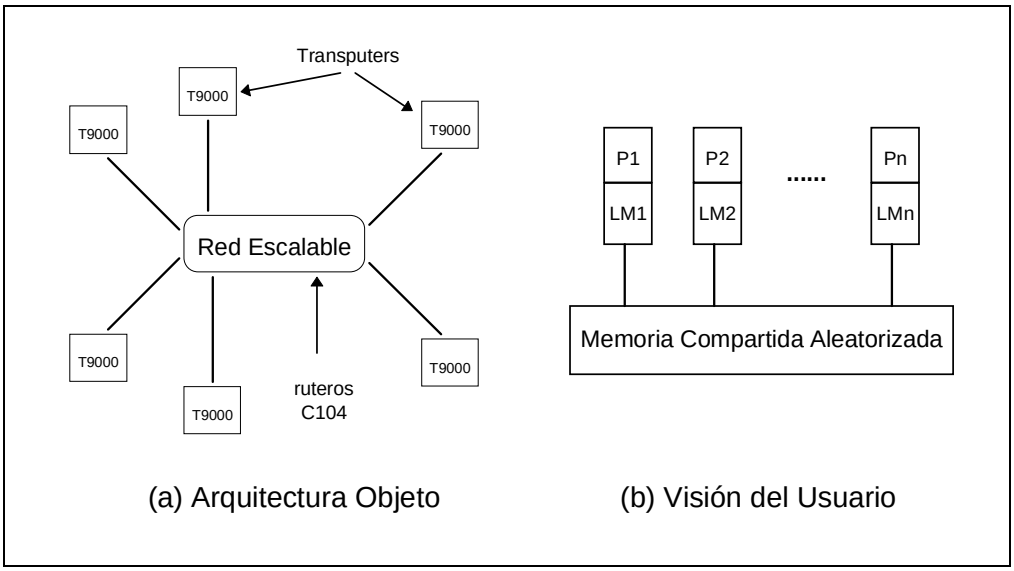


Figura 5. Simulador de la WPRAM.

Los tres modelos: BSP, LogP y WPRAM tienen en común una estructura de memoria en dos niveles con acceso global uniforme. A pesar de las diferencias, como el número y significado de los parámetros, el análisis del rendimiento en esos modelos resulta muy similar cuando se utiliza sincronización por barrera dentro de los algoritmos a estudiar.

3. Extensiones del Modelo PRAM.

A diferencia de los modelos introducidos en la sección precedente , los modelos de esta sección aumentan la PRAM básica añadiéndole nuevas características hardware y/o un mayor juego de instrucciones. La incorporación de nuevas instrucciones de cálculo paralelo tiene como objetivo simplificar la escritura de algoritmos para determinadas clases de problemas. Esto se hace con la esperanza de que el costo del hardware suplementario introducido sea compensado por lo que se gana en eficacia.

3.1 Memoria inalterable.

Vishkin propone en [Vis85] una variante del modelo PRAM en la cual a la memoria compartida con *M* palabras se le añade una memoria compartida inalterable ROM (Read-Only Memory). La ROM contiene los *n* datos del problema. El uso de la ROM añade más potencia sólo en el caso en que $M = O(n)$.

3.2 Scan.

El modelo PRAM supone que el tiempo de acceso a memoria es constante, aún cuando para toda implementación del modelo el tiempo de acceso deberá ser una función del tamaño de la memoria compartida (en general será exactamente del orden de *log M* para una memoria con *M* direcciones). Sea $\oplus$ una operación binaria, *i* un elemento neutro y $[a_0,a_1,...,a_{n-1}]$ un vector. La operación prefijo calcula el vector $[i,a_0, a_0 \oplus a_1,...,a_0 \oplus a_1 \oplus a_2 ... \oplus a_{n-2} \oplus a_{n-1}]$. Varias funciones de gran utilidad, tales como la difusión pueden ejecutarse utilizando la operación de prefijo. Nótese que esta última puede calcularse en tiempo *log n*, es decir, en un tiempo igual al tiempo de un acceso a memoria, salvo una constante. Esta observación motiva el modelo Scan que fue introducido por [Ble89] y que es esencialmente una EREW PRAM aumentada con una operación de prefijo, considerando que estas últimas requieren tiempo constante. Con la ayuda de estas operaciones, se simplifica la descripción de numerosos algoritmos PRAM. Además, para ciertos algoritmos EREW PRAM y ciertos algoritmos CRCW PRAM el tiempo puede reducirse por un factor *log n*.

3.3 BSR .

El modelo BSR (Broadcasting with Selective Reduction) debido a [Akl89b] es una CRCW PRAM aumentada con una instrucción DIFFUSION que puede ser ejecutada en un tiempo igual al del tiempo de un acceso a memoria compartida. Sea *R* la operación de reducción, una operación

binaria como la suma, el producto, el máximo, el mínimo, el AND, OR, XOR. Sea σ la operación de selección tal como <, <=, =, >=, >, <>. Cada celda de memoria compartida m_j recibe un valor límite *l[j]*. Cada procesador *p[i]* especifica un dato *d[i]* y una etiqueta *t[i]*. En la ejecución de una instrucción DIFFUSION, la celda *m[j]* acepta todos los datos *d[i]* tales que cumplen *t[i]σl[j]*. Todos los datos aceptados por la celda de memoria son combinados por *R*, y el resultado se almacena en *m[j]*. Por lo tanto es posible que un procesador escriba simultáneamente en todas las celdas de la memoria compartida. Así mismo, toda celda puede recibir los datos de todos los procesadores a la vez. La DIFFUSION es una primitiva bastante potente que simplifica al mismo tiempo la concepción y descripción de varios algoritmos PRAM. Por ejemplo, una única DIFFUSION por *N* procesadores puede calcular un prefijo. La implementación de esta instrucción puede hacerse utilizando los mismos recursos que requiere un acceso de escritura concurrente y en el mismo tiempo que consume tal acceso.

3.4 H-PRAM.

El modelo H-PRAM (Hyerarchical-PRAM) introducido en [Hey91] dispone de una instrucción PARTITION que permite dividir una PRAM de tipo SIMD síncrona en un número de PRAMs SIMD síncronas disjuntas, que operan entre sí de manera asíncrona. A su vez, cada sub-PRAM puede dividirse, dando a la H-PRAM una jerarquía de PRAMs en forma de árbol.

3.5 PRAM asíncronas.

En la literatura también se pueden encontrar modificaciones del modelo PRAM que operan en modo MIMD, no proporcionan sincronización automática entre procesadores y reflejan algunas de las implementaciones existentes de memoria compartida.

Uno de tales modelos es descrito por P. B. Gibbons en [Gib89a, Gib89b] que comprende todas las formas de acceso a memoria, así como una operación de sincronización que se aplica bien a conjuntos distintos de

procesadores, bien a todos los procesadores. El tiempo requerido por esta operación es una función no decreciente B(p) del número de procesadores implicados p. Estos últimos pueden operar a diferentes velocidades y son el objeto de ciertos retrasos finitos. Un algoritmo para este modelo debe funcionar perfectamente a pesar de estos retrasos. Además, un procesador que desea hacer una lectura de una celda de memoria debe ser sincronizado con el procesador que realizó la última escritura, de manera que hay sincronización después de la escritura y antes de la lectura. Se ha demostrado en [Nis90] que esta restricción debería causar una degradación en el rendimiento de ciertos algoritmos, tales como los que utilizan el método del salto de punteros. Un programa para este modelo es considerado como una serie de fases separadas por etapas de sincronización. Si concebimos que los algoritmos tengan O(B(p)) etapas por fase, se alcanza un equilibrio entre los costes de cómputo y sincronizaciones. Un algoritmo para este modelo se analiza suponiendo ausencia de retrasos. Se puede también introducir un parámetro para la latencia de memoria, aunque se supone que las demandas de accesos a memoria durante una fase se tratan mediante segmentación (*pipeline*).

Un segundo modelo PRAM asíncrono es el descrito en [Col89, Col90]. En este modelo las lecturas y escrituras concurrentes están permitidas, y los procesadores operan a velocidades diferentes y variables. El tiempo requerido por una etapa de un algoritmo se elige de manera aleatoria. Un algoritmo para este modelo debe operar correctamente independientemente de las velocidades de los procesadores. Los flujos de instrucciones de los diversos procesadores se intercalan según una distribución de probabilidad. Al analizar un algoritmo, el tiempo de cálculo es igual a la media del tiempo máximo requerido por un procesador. Esta elección tiene por objetivo reflejar el tiempo probable sobre una máquina real.

4. Propuestas de Realización.

Aunque se han publicado varias propuestas de implementación (simulaciones o emulaciones) del modelo PRAM, ninguna puede considerarse realizada de manera efectiva hasta el momento, quedándose en el mero plano teórico. Las implementaciones pueden clasificarse en: deterministas, probabilistas, uniformes y no-uniformes.

4.1 La Simulación de Vishkin.

En [Vis84] Vishkin propone la implementación de una CRCW PRAM con *p* procesadores con la ayuda de un grafo. Los procesadores y las celdas se colocan a la entrada de un retículo. Si se utiliza el retículo descrito en [Ajt83] la implementación propuesta requiere *O(p log p + m log m)* nodos de procesamiento y un tiempo de *O(log p + log m)* por acceso a memoria. Esta implementación se considera uniforme y determinista.

4.2 La Simulación de Hagerup.

La implementación de la PRAM propuesta en [Alt87], incluye entre sus autores al profesor Hagerup que estuvo vinculado, en sus comienzos, al diseño del lenguaje orientado al modelo PRAM FORK y que es el objeto de estudio del capítulo 2 del presente trabajo. De la misma forma que en otras numerosas realizaciones, *p* procesadores estarán conectados mediante un grafo de grado constante. A cada procesador se le asocia un módulo de memoria con *M/p* celdas. Aquí *M=O(mlog(m))*, donde *m* denota el número de celdas de la memoria compartida de la PRAM simulada. Cada variable común se representa por $2\lfloor \log m \rfloor - 1$ copias. Con el fin de asegurar la actualización con éxito del valor de una variable, el nuevo valor así como una etiqueta deben ser almacenadas en una mayoría $\lfloor \log m \rfloor$ de copias de la variable. Por consiguiente, una lectura termina cuando se encuentran $\lfloor \log m \rfloor$ copias iguales y con la misma etiqueta. Cada acceso a memoria necesita

un tiempo de orden *O(log n log m)* con $n < m \leq 2^n$. Esta implementación es determinista, pero no uniforme.

La implementación de Ranade que se describe a continuación es no-uniforme y probabilista y es la base para la construcción de la máquina SB-PRAM desarrollada en Saarbrücken.

4.3 El modelo de Ranade y la SB-PRAM.

La Máquina Fluida (*Fluent Machine*) de Ranade [Ran89a, Ran89b] utiliza una red de interconexión con topología de mariposa con $N = (n+1)\times 2^n$ conmutadores, procesadores y módulos de memoria. Los enlaces son bidireccionales. Una topología de mariposa con *n+1* columnas o etapas se define como un grafo G = (V,E) con $V = \{0,...,n\} \times \{0,..., 2^n-1\}$. Para *0 <= i < n*, el Nodo <i, x> está conectado con los nodos $\langle i+1, x\rangle$ y $\langle i+1, x \oplus 2^i\rangle$. Aquí, $a \oplus b$ denota la representación binaria de un entero obtenida mediante una operación or-exclusiva de las representaciones binarias de los enteros *a* y *b*. Se denominará a los nodos $\langle 0,x\rangle,..., \langle n,x\rangle$ una *fila*, y a los nodos $\langle i,0\rangle,..., \langle i, 2^n-1\rangle$ una *columna* o *etapa*.

El espacio de direcciones tiene tamaño *m*. La distribución de los módulos de memoria está dada por una función de la forma:

$$h(x) = \sum_{i=0}^{O(\log m)} a_i x^i \bmod P \bmod N$$

Donde *P* es un primo mayor que *m*. Se escoge aleatoriamente una función particular para elegir los coeficientes a_i entre *0* y *P-1*. Como cada función distribuye únicamente muy pocos patrones de acceso de tal manera que uno de los módulos se quede sobreagotado por las peticiones, la función de hash elegida distribuirá bien el tráfico a memoria para un programa de aplicación dado con una probabilidad alta.

Para una descripción simple del algoritmo de encaminamiento, considererese una red lógica que conste de 6 redes de mariposa como la que se muestra en la Figura 6. La máquina opera en 6 fases:

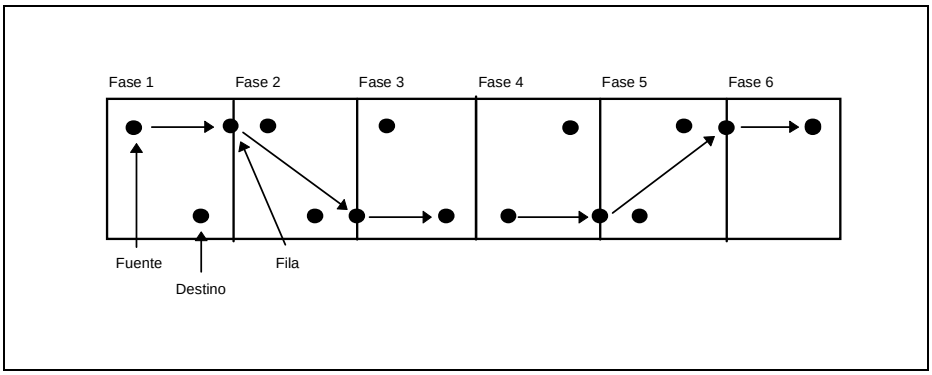


Figura 6. La Máquina Fluida de Ranade

Al principio de la primera fase, todos los procesadores introducen sus peticiones en la red. Las peticiones constan de la dirección hash de la celda de memoria que debe ser accedida, del tipo del acceso (lectura o escritura) y de, en el caso de una escritura, del dato. Durante la primera fase, todas las peticiones (o paquetes) de una fila son desplazados hasta el final de la fila y ordenados según direcciones hash. Durante la segunda fase, cada petición es encaminada a su fila destino. En la fase número 3, las peticiones son desplazadas hacia sus columnas destino donde tienen lugar los accesos a memoria. Un conmutador en esta fase tiene una entrada y dos salidas, una al modulo de memoria y una al conmutador en la misma fila y la siguiente columna. En las tres últimas fases, cada petición recorre su camino en la dirección inversa y regresa al procesador que inició la petición.

Durante la fase número dos, las peticiones son ordenadas. Si los dos buffers de entradas de un conmutador de encaminamiento contienen paquetes, procederá aquel que tenga menor dirección hash. Si las dos direcciones son idénticas, los dos paquetes se combinan en uno. La ordenación garantiza que todos los paquetes destinados a la misma dirección sean conocidos y combinados en uno. La idea de la ordenación puede introducir tiempos de espera adicionales.

Cada conmutador que transmite una petición por una salida, envia un mensaje fantasma por la otra. El mensaje fantasma acarrea la misma dirección, pero tiene un tipo especial GHOST y no contiene datos. Si el tope de cada buffer de entrada contiene un paquete que está esperando para ser trasmitido el control del conmutador selecciona siempre el paquete con menor destino. Entonces el paquete fantasma será el primero en ser trasmitido. Un paquete de lectura o escritura en la columna i puede ser seleccionado sólo si todos los i fantasmas que origina la columna i han pasado antes. El (i+1)-ésimo paquetes que llega a <col, row> es el que tiene el menor destino entre los paquetes en las columnas 0,...,i-1 de la fila row.

Las fases de la cuatro a la seis, no toman decisiones de encaminamiento. Las decisiones de las fases primera a la tercera son almacenadas en las “colas de direcciones” de cada conmutador y son utilizadas para controlar el comportamiento de los mismos en las restantes fases.

Ranade demuestra en [Ran89a] que con la anterior función de hash y con el anterior algoritmo de rutas, un paso de la PRAM puede emularse en tiempo *b log n* con una probabilidad muy grande, caundo los buffers de los conmutadores pueden guardar *b* paquetes.

En los módulos de memoria de la Máquina Fluida surge el problema del hashing secundario. La solución esbozada en [Ran89b] amplía el tamaño de los módulos por un factor constante.

Una desventaja del diseño de Ranade es que los procesadores están ociosos la mayor parte del tiempo. Para mejorar esto sería necesario establecer una segmentación (pipeline) de las fases del encaminamiento. En el diseño de la SB-PRAM, se utilizan dos redes de mariposa que realizan las fases dos y cinco descritas en los párrafos anteriores, utilizan un algoritmo de ordenación de arrays para las fases uno y seis, y usan 2^n módulos con bancos múlitples para obviar las fases tres y cuatro. Una descripción detallada de los cambios realizados se encuentra en [Abo91a,

Abo91b]. Cada procesador de una fila se materializa en un procesador físico en el que corren *cn* procesadores virtuales en una segmentación. Obtienen un total de $p = 2^n$ procesadores físicos y $N = cnp$ procesadores virtuales. Cada procesador virtual tiene su propio conjunto de registros en hardware. El juego de instrucciones es similar al procesador RISC Berkeley, cada instrucción consume la misma cantidad de tiempo. La función hash propuesta por Ranade también se cambia en la SB-PRAM, puesto que una evaluación polinomial de orden *O(log m)* es lenta y costosa en términos de hardware. Los autores utilizan una función lineal de la forma $h(x) = ax \bmod m$.

Capítulo II

Lenguajes Orientados al Modelo PRAM

1. Lenguajes Orientados al Modelo PRAM.

Puesto que muchos investigadores que trabajan en el diseño de algoritmos usan el modelo de computación PRAM, el número de algoritmos publicados para el modelo PRAM es grande y continúa creciendo. Esto se debe principalmente al poderoso mecanismo de comunicación que representa la memoria global compartida. Curiosamente, no hay un lenguaje de programación estándar para el modelo PRAM y cada diseñador que intente formalizar un algoritmo para este modelo desarrolla su notación particular. Entre las desventajas de esta política se podría citar que al menos potencialmente, las dificultades de expresión y comprensión se ven agravadas por la carencia de un lenguaje común; además las mismas

definiciones se repiten una vez y otra. Como el diseñador del algoritmo está más interesado en el algoritmo que en la notación utilizada para describirlo, cualquier fragmento de lenguaje que pudiera introducir tiene pocas posibilidades de ser actualizado de manera estándar en el diseño de un lenguaje de programación.

Sin embargo, en la computación en paralelo se ha hecho un gran esfuerzo en desarrollar lenguajes de programación adecuados. La mayoría de esos lenguajes se han pensado para ser usados con sistemas multiprocesador consistentes en computadores autónomos, cada uno con su propio reloj, que corren principalmente de forma independiente, y ocasionalmente intercambian mensajes. La comunicación y la sincronización entre procesadores pueden estar basadas en el paso de mensajes (Ada [Uni83], Occam [Inm91], Concurrent C [Geh88]), o bien en variables compartidas protegidas (Concurrent Pascal [Han75]). Estos lenguajes no soportan una memoria global accesible simultáneamente por todos los procesadores. Mientras que tales lenguajes son excelentes herramientas en la computación distribuida, no son adecuados para la descripción de algoritmos PRAM.

Otra aproximación para introducir paralelismo en los lenguajes consiste en ampliar lenguajes secuenciales que se ejecutan en procesadores ordinarios con primitivas de comunicación que permitan el acceso a variables compartidas. Se han propuesto e implementado varias librerías con funciones para estos propósitos sobre distintas máquinas paralelas, especialmente para extender los lenguajes C [Ans89] y FORTRAN [Ame78].

Se han hecho intentos de diseñar lenguajes síncronos para el paradigma del Paralelismo de Datos (data-parallel). Este tipo de computación surge frecuentemente en cálculo numérico. El paralelismo de datos consiste principalmente en la ejecución paralela de iteraciones sobre matrices de grandes dimensiones. Los lenguajes imperativos de paralelismo de Datos se han diseñado especialmente para programar máquinas SIMD,

esto es, procesadores vectoriales segmentados o *arrays* de procesadores como la CM2. Como ejemplos de tales lenguajes se pueden citar Vector C [Li85] y C* [Ros87] o sus parientes Data parallel C [Hat91] y DBC [Sch92] y los recientes HPF [Lov93] y Modula 2* [Tic90]. Todos estos lenguajes admiten un espacio de nombres globlal, pero no soportan otros paradigmas de computación como un estilo divide y vencerás paralelo y recursivo como los que se sugieren en [JáJ92].

La idea básica detrás del paralelismo de datos en Fortran 90 [Met90] es permitir expresiones en las cuales la aparición del nombre de un array implique que todos sus elementos son procesados concurrentemente. El lenguaje Fortran D es una extensión de Fortran que capacita al usuario a especificar la descomposición de su área de datos. También soporta un bucle *forall* que indica al compilador que ha de asegurarse de que no existen dependencias entre las distintas iteraciones. Por ejemplo, si una iteración escribe en una variable que la otra lee, se garantiza que el valor leído es el antiguo, esto es, el valor en el momento en el que se entró en el bucle. Otra característica del Fortran D es que permite especificar en qué procesador se ejecutará cada iteración.

Un lenguaje orientado al modelo PRAM ha de contar con características que permitan escribir programas para una máquina de memoria compartida. El lenguaje debería distinguir entre variables compartidas, las cuales pueden ser accedidas por un cierto grupo de procesadores y variables privadas de las que pueden existir varias instancias, cada una de las cuales reside en la memoria privada de un procesador diferente y posiblemente tengan distintos valores. El lenguaje también debería reflejar la capacidad de la máquina de acceder de manera síncrona a los datos compartidos. Finalmente, la admisión de recursividad por parte de los lenguajes dota a los programas secuenciales de una escritura clara y bien estructurada, por ello deberían poderse combinar libremente con el paralelismo. La recursividad se caracteriza por subdividir un problema dado en un conjunto de subproblemas que se pueden resolver

independientemente y posiblemente en paralelo. Cada subproblema se puede trabajar nuevamente sobre varios procesadores. El lenguaje de programación debería proporcionar al programador la forma de generar subgrupos de trabajo que se ejecuten síncronamente sobre los procesadores.

Los lenguajes Modula2* [Tic90] y pm2 [Ham92, Juv92a, Juv92b] aunque no establecen diferencias entre variables privadas y compartidas y carecen de la capacidad de combinar recursividad y paralelismo, ofrecen una semántica síncrona acorde con el modelo PRAM.

El lenguaje de programación pm2 es un superconjunto de Modula2 [Nik85] mezclado con Pascal. Las principales diferencias con Modula2 son los bucles paralelos (sentencias *par*), la inicialización y finalización de los procedimientos y la ausencia de módulos. En pm2 existen dos formas de sincronizar procesadores, una sincronización implícita y otra explícita. Si se le añade la opción *sync* a la sentencia *par* se provoca que todos los procesadores la ejecuten síncronamente. La sentencia *synchronize* provoca que todos los procesadores activos se sincronicen en ese punto (sincronización por barrera).

Modula2* es una extensión de Modula2 que permite escribir programas paralelos portables a distintas arquitecturas. Las extensiones hacen abstracción de la organización de la memoria y del número de procesadores físicos disponibles. El paralelismo es explícito y el programador puede escoger entre ejecución síncrona y asíncrona. Por lo tanto, los programas pueden utilizar el modelo SIMD donde abundan las sincronizaciones o utilizar los modelos MSIMD o MIMD donde las sincronizaciones son poco frecuentes.

Las extensiones en Modula2* consisten en versiones síncronas y asíncronas de la sentencia *forall*, además de declaraciones opcionales para la asignación de matrices sobre procesadores de forma independiente de la máquina. En el lenguaje no se hace visible una red de interconexión entre los procesadores. Se asume un espacio de direcciones compartido entre todos los procesadores, no tiene que ser necesariamente memoria compartida. No hay

sentencias explícitas para el paso de mensajes, su uso se evita mediante la lectura o escritura en posiciones del espacio de direcciones compartido. El programador puede influir en la distribución de datos mediante declaraciones, pero éstas son sólo ayudas para el compilador que no tienen ningún efecto sobre la semántica de un programa.

La sentencia *forall* crea un conjunto de procesos que se ejecutan en paralelo. En su forma asíncrona, los procesos individuales operan concurrentemente y se pueden sincronizar explícitamente con sentencias SIGNAL y WAIT. La sentencia *forall-asíncrona* termina cuando termina el último de de los procesos creados. En su forma síncrona, los procesos creados por la sentencia *forall* operan al unísono hasta que alcanzan un punto de ramificación tal como una sentencia *if* o una sentencia *case*. En los puntos de ramificación, un conjunto de procesos puede dividirse en dos o más subconjuntos. Los procesos dentro del mismo subconjunto continúan operando al unísono, pero los subconjuntos no se sincronizan uno con respecto al otro. Por lo tanto, se puede decir que la unión de los subconjuntos se comporta del modo MSIMD. Una sentencia que provoca una división en subconjuntos termina cuando terminan todos sus subconjuntos, en ese punto los subconjuntos se unen para continuar con la ejecución de la siguiente sentencia. La sentencia *forall-síncrona* es con frecuencia más fácil de manipular que la asíncrona puesto que ésta evita sincronizaciones peligrosas. Sin embargo la forma síncrona puede ser extremadamente restrictiva y podría llevar a no utilizar todo el potencial de la máquina. La combinación de la formas síncronas y asíncronas en Modula-2* permite la programación paralela de una gran cantidad de estilos entre SIMD y MIMD.

Actualmente, sólo conocemos la existencia de dos lenguajes que cumplan todos los requisitos especificados para un lenguaje orientado al modelo PRAM: FORK y ll. A continuación se abordará el estudio del lenguaje FORK y de la única y más reciente implementación del mismo, el lenguaje *fork95*. El lenguaje ll se presentará en el capítulo siguiente.

2. El Lenguaje FORK.

FORK [Hag92] es un lenguaje basado en Pascal, en el que se pueden expresar algoritmos PRAM con un alto nivel de abstracción. Los lenguajes de propósito general tales como Ada, Occam, o Concurrent C no son adecuados para expresar algoritmos PRAM, puesto que ninguno de ellos soporta adecuadamente el concepto de variables compartidas alojadas en la memoria compartida, ni la ejecución de programas con sincronizaciones en cada sentencia. En FORK se pueden usar dos clases de variables: las variables *privadas* se corresponden con la memoria local del procesador y las *compartidas* con la memoria global. El *índice de un procesador* se reemplaza por un número de procesador lógico, que el usuario puede definir, y las instrucciones PRAM se reemplazan por sentencias tipo Pascal.

Varios procesadores pueden ejecutar de forma síncrona una sentencia, pero tan pronto como dos procesadores ejecuten sentencias diferentes se asume que se ejecutan de manera asíncrona. Este es un requisito necesario puesto que la ejecución síncrona de sentencias diferentes requiere un conocimiento exacto de las instrucciones PRAM a las que da lugar una sentencia, y esto destruye la abstracción que se pretende conseguir.

Los conflictos de escritura pueden aparecer durante la ejecución síncrona y se resuelven mediante alguna estrategia asociada a la CRCW PRAM. En este sentido FORK es una *clase* de lenguajes de programación y hay un representante para cada modelo CRCW.

FORK soporta la combinación de paralelismo y recursividad. La recursividad se caracteriza por una subdivisión de un problema dado en un conjunto de subproblemas, que se pueden resolver independientemente y posiblemente en paralelo. Cada subproblema lo pueden trabajar varios procesadores (un grupo de procesadores). La resolución de manera recursiva y paralela de problemas conlleva la organización de los grupos

de procesadores en una jerarquía. FORK posibilita al programador la distinción entre datos compartidos y datos privados. Los datos compartidos lo son siempre relativos a un grupo de procesadores, lo que significa que pueden ser accedidos por todos los procesadores dentro de ese grupo o un subgrupo (con respecto a la jerarquía), pero no por procesadores externos al grupo.

Las características más destacadas de FORK son las siguientes:

- Se añade una sentencia *start*, que permite inicializar un conjunto de procesadores, con índices en el rango especificado.
- Se realiza la extensión semántica que garantiza la ejecución síncrona del programa (y también los accesos síncronos a los datos).
- Se asume una sincronización implícita al comienzo de cada sentencia.
- Se introduce una división implícita en subgrupos en cada punto de ramificación del programa donde la ramificación que tiene lugar en un procesador depende de sus variables privadas.
- Se asume que las sentencias disponibles en el lenguaje se pueden anidar libremente. En particular las iteraciones y recursividad son compatibles tanto con la inicialización de nuevos procesadores como con la división en subprocesos.

El paralelismo en FORK está controlado mediante dos sentencias: *start[<expr>..<expr>]* y *fork[<expr>..<expr>]*. La sentencia *start* se usa para reajustar el grupo de procesadores disponible para la ejecución de un programa PRAM, mientras que la sentencia *fork* no cambia el número de procesadores, sino que crea subgrupos que operan independientemente sobre distintas subtareas y permite una distribución precisa de los procesadores disponibles sobre las tareas. El efecto de estas sentencias junto con el concepto de FORK de ejecución síncrona de un programa se explican con detalle a continuación.

2.1 La Semántica de FORK.

Un concepto básico de FORK es el de procesador lógico. Los procesadores lógicos se proyectan en los procesadores físicos que proporciona la arquitectura. Sin embargo, el número de procesadores lógicos que están trabajando durante la ejecución de un programa puede variar, así, el número de procesadores lógicos puede exceder el número de procesadores físicos disponibles. Cada procesador lógico *p* posee una constante entera que lo distingue, cuyo valor se refiere al *número del procesador* p y que se denota por el símbolo #. También puede tener otras constantes, tipos y variables privadas que son accesibles sólo por él. Los datos declarados como compartidos por un grupo de procesadores pueden ser accedidos por todos los procesadores del grupo.

FORK permite combinar paralelismo y recursividad. Esto hace surgir el segundo concepto básico de FORK: la idea de *grupo*. Un grupo está formado por un conjunto de procesadores (sin excluir el vacío). Las variables compartidas lo son siempre relativas a un grupo de procesadores, lo que significa que los procesadores dentro del grupo pueden acceder a ellas pero no los procesadores que están fuera de él. Los grupos se pueden dividir en subgrupos. Los grupos establecidos más recientemente reciben el nombre de *grupos hoja*. Los grupos hoja juegan un papel especial en FORK: los procesadores dentro de un grupo hoja trabajan síncronamente. Si se declaran nuevos datos compartidos, se establece que son compartidos en relación al grupo hoja que está ejecutando la declaración. Cada grupo tiene un *número de grupo*. El número del grupo creado más recientemente puede ser accedido por los miembros del grupo a través de la constante privada @. El valor de @ será el mismo para todo el grupo. Inicialmente sólo hay un grupo con número de grupo 0 que consta del procesador con número de procesador 0.

Los programas FORK son ejecutados por procesadores que están organizados mediante una jerarquía de grupos. Para cada sentencia de

FORK se explicará cómo su ejecución afecta a la jerarquía de grupo, a la sincronización entre procesadores y al ámbito de los datos.

Se utilizará la siguiente terminología: Una *jerarquía de grupo* H es un árbol finito cuyos nodos son conjuntos de procesadores. Un nodo de H se denomina un *grupo* (de H). Asumamos que G es un grupo. Un *subgrupo* de G es un nodo en el subárbol con raíz G, y un *grupo hoja* de G es una hoja de este árbol. Un procesador *p* pertenece a un grupo G si *p* pertenece a un grupo hoja de G. Todos los procesadores en un grupo síncrono maximal están en el mismo punto del programa. Cada grupo hoja es un subgrupo de un grupo síncrono maximal. Un grupo G se dice síncrono si es un subgrupo de un grupo síncrono maximal. Antes de la ejecución de un programa, la jerarquía de grupo H consta sólo de un grupo etiquetado con el 0 que contiene sólo un procesador con número 0. Este grupo es síncrono maximal. Al final de la ejecución del programa se tiene nuevamente esta jerarquía H. Las secuencias de declaraciones y sentencias son ejecutadas por grupos síncronos maximales (con respecto a la jerarquía de grupo actual). En los párrafos que siguen, G siempre denota un grupo síncrono maximal con respecto a la jerarquía de grupo actual y H el subárbol con raíz G. La ejecución de declaraciones y sentencias puede cambiar la jerarquía de grupo y las sincronizaciones, pero sólo dentro del subárbol H.

2.1.1 Declaraciones.

Asumamos que G ejecuta una secuencia de declaraciones como la siguiente:

<Declaración> **;** <Decls>

Entonces, G primero ejecuta <Declaración>. Durante la ejecución de <Declaración> la sincronización entre los procesadores que la ejecutan y la jerarquía de grupo con raíz G puede cambiar. Sin embargo, al final de

<Declaración> , se restablece H, y G es síncrono maximal nuevamente con respecto a la jerarquía de grupo actual.

A continuación, G empieza a ejecutar <Decls>. Si un dato es privado, se crean objetos en cada procesador que ejecute la declaración. Si el dato se declara como compartido, cada grupo hoja ejecuta esta declaración y tiene un objeto distinto el cual es accesible por todos los procesadores de ese grupo hoja.

2.1.2 Sentencias.

A nivel de máquina cada instrucción consume exactamente una unidad de tiempo. Sin embargo, la semántica de un lenguaje de alto nivel debería ser independiente de las características especiales del esquema de traducción. Por lo tanto, no se debería especificar cuantas unidades de tiempo se consumen por ejemplo en una sentencia de asignación. Por esta razón se hace necesaria una noción de *Ejecución Síncrona de Programa* que sólo dependa de texto del programa en sí. Esta noción no debería confundirse con la noción de reloj global de una PRAM física; esto es, el hardware podría permitir que diferentes procesadores ejecuten diferentes instrucciones durante el mismo ciclo de reloj, mientras que la noción de sincronización no permite una ejecución síncrona de sentencias diferentes.

Ser síncronos es una propiedad de un conjunto de procesadores. Esto implica que todos los procesadores dentro de ese conjunto están en el mismo punto del programa. Esto significa que ejecutan la misma sentencia dentro del mismo bucle dentro del mismo procedimiento y además el contexto de las llamadas recursivas a procedimientos y el número de iteraciones dentro del mismo bucle coinciden. No existen mecanismos de sincronización explícita en FORK. Las sincronizaciones implícitas en FORK se hacen sentencia a sentencia. Al final de cada sentencia hay un punto de sincronización implícito. Esto significa que si un conjunto de procesadores ejecuta síncronamente una secuencia de sentencias de la forma:

$$\text{<Sentencia>}_1 \ ; \ \text{<Sentencia>}_2$$

los procesadores del conjunto ejecutan en primer lugar <*Sentencia*>$_1$. Cuando todos los procesadores han finalizado la ejecución de <*Sentencia*>$_1$ ejecutan síncronamente <*Sentencia*>$_2$. Nótese que dentro de la ejecución de <*Sentencia*>$_1$ procesadores diferentes pueden alcanzar diferentes puntos de programa. Entonces pueden convertirse en asíncronos dentro de <*Sentencia* >$_1$.

FORK es un lenguaje estructurado y no tiene sentencias *goto*. De esta forma no se pueden obviar los puntos de sincronización implícitos. Por lo tanto, no es posible una espera infinita por parte de los procesadores.

Sea G es un grupo síncrono maximal que ejecuta una secuencia de sentencias de las forma:

<*Sentencia*> **;** <*Sents*>

Entonces, G primero ejecuta <*Sentencia*>. Durante la ejecución de <*Sentencia*> la sincronización entre los procesadores que la ejecutan y la jerarquía de grupo H con raíz G puede cambiar. Sin embargo, al final de <*Sentencia*>, H se restablece, y G es nuevamente síncrono maximal con respecto a la jerarquía de grupo actual. G comienza entonces la ejecución de <*Sents* >.

2.1.2.1 Creación de Nuevos Procesos: La Sentencia start.

La sentencia *start* se utiliza para activar procesadores. Los nuevos procesadores no tienen acceso a los datos privados de los antiguos procesadores. Por lo tanto, las sentencias dentro de la sentencia *start* no deberían referirse a tipos, variables o constantes privadas, declaradas fuera de la sentencia *start*, excepto a @ y #.

Cuando los procesadores de G ejecutan la sentencia:

---

start *[*< *expr* >$_1$..< *expr* >$_2$*]* <*Sents*> ***endstart***

En primer lugar evalúan el rango $< expr >_1 .. < expr >_2$. Este rango no debe depender de variables privadas. Por lo tanto el rango obtenido por todos los procesadores de un grupo hoja *g* es el mismo: $v_{g,1},\ldots,v_{g,2}$. Se añade a cada grupo hoja *g* de G un nuevo grupo hoja que contiene $v_{g,2} - v_{g,1} + 1$ nuevos procesadores numerados con los elementos de $\{v_{g,1},\ldots,v_{g,2}\}$. El número de grupo (@) de los nuevos grupos hoja es el mismo que el de su grupo padre. G continúa siendo síncrono maximal. Ahora G ejecuta <*Sents*>, lo que significa que los procesadores nuevos de los grupos hoja nuevos ejecutan <*Sents*> síncronamente. Cuando G alcanza *endstart*, los grupos hoja son eliminados, esto es, se restablece la jerarquía H original.

Consideremos un bosque F con N elementos. Se representa a F mediante un array de enteros, A, donde A[i] = i es la raíz de un árbol de F y A[i] contiene al padre de i en otro caso. Fijada una constante entera N, en el Código 1 se propone una solución al problema de encontrar la raíz del árbol en F al cual pertenece un nodo i, con i en en el rango 1..N. Los valores se almacenan en un array R. Como en Pascal, la constante N, la variable del bucle t y los arrays A y R se deben declarar en el ámbito correspondiente. En FORK estas declaraciones deben indicar además si la variable es compartida o privada.

Inicialmente sólo trabaja un procesador con número de procesador 0. La sentencia start[1..N] de la línea 5 inicializa los procesadores con números (#) 1,…,N. La correspondiente sentencia *endstart* detiene a estos procesadores y restablece nuevamente a los procesadores activos anteriormente. Así pues, una sentencia start[1..N] seguida inmediatamente por una sentencia start[1..M] no inicializa N×M procesadores sino sólo M procesadores. Cuando tiene lugar la sentencia *endstart* finaliza la fase en la que tienen que trabajar M procesadores y se restablecen N procesadores con números 1,…,N.

```
        ...
( 1)    shared cosnt N=...;
( 2)    shared var t: integer;
( 3)    shared var A: array[1..N] of integer;
( 4)    shared var R: array[1..N] of integer;
        ...
( 5)    start [1..N]
( 6)      R[#] := A[#];
( 7)      for t := 1 to log(N) do      /* log (N) denota la parte entera superior de log2 (N)*/
( 8)        R[#] := R[R[#]];
( 9)      enddo
(10)    endstart
        ...
```

Código 1. Ejemplo de una Sentencia start

2.1.2.2 Formación de Grupos de Procesadores: La Sentencia fork.

La sentencia fork se utiliza para generar de forma explícita nuevos grupos hoja. A los nuevos grupos hoja se les asignan nuevos números de grupo y se renumeran los procesadores dentro de los nuevos grupos hoja.

Asumamos que los procesadores de G ejecutan la sentencia:

fork [$< expr >_1 .. < expr >_2$]
@ = $< expr >_3$;
# = $< expr >_4$;
<Sents>
endfork

Primero, cada procesador p de G evalúa las expresiones $< expr >_1, ..., < expr >_4$. Las expresiones $< expr >_1$ y $< expr >_2$ no deben depender de datos privados, por lo que todos los procesadores dentro del mismo grupo hoja g de G evalúan $< expr >_1$ y $< expr >_2$ al mismo valor $v_{g,1}$ y $v_{g,2}$ respectivamente. En cada grupo hoja g de G se añaden $v_{g,2} - v_{g,1} + 1$ nuevos grupos hojas, los cuales son numerados con los elementos de $\{v_{g,1}, \ldots, v_{g,2}\}$. El subconjunto de procesadores de g que evalúan $< expr >_3$ a el valor "*i* " etiquetan al nuevo grupo hoja con dicho valor. Cada uno de estos procesadores obtiene el valor de $< expr >_4$ como su nuevo número

de procesador. Nótese que pertenecen al mismo grupo aquellos procesadores de G que evalúan $< expr >_3$ al mismo valor $v_{g,3}$. El número de evaluaciones distintas $v_{g,3}$ de $< expr >_3$ debe coincidir con $v_{g,2} - v_{g,1} + 1$, lo contrario produce un error en tiempo de ejecución. G permanece síncrono maximal y ejecuta <Sents>. Cuando G ejecuta el *endfork*, los nuevos grupos hoja son eliminados, esto es, se restablece la jerarquía de grupo original.

Consideremos como ejemplo el algoritmo genérico divide y vencerás del Código 2. El parámetro N de DC describe el número máximo de procesadores disponibles para resolver el problema dado y los parámetros adicionales contienen los datos necesarios. Se asume además que cada paso de la recursividad divide el problema en $\sqrt{N}$ subproblemas de tamaño $\sqrt{N}$. Los valores de # de los procesadores en los grupos hoja que llaman a DC no forman necesariamente un intervalo continuo. Esta situación no se produce en el algoritmo alternativo que se presenta en el Código 3. Pueden existir subtareas a las que se les asigna un conjunto de procesadores vacío y no ejecutan nada.

```
procedure DC( shared const N: integer; …);
…
if trivial(N) then conquer(…)
else
  fork [0..sqrt(N)-1]
  @ = # div sqrt(N);
   # = # mod sqrt(N);
     DC( sqrt(N), …)
   endfork
endif;
…
```

Código 2. Un Algoritmo Genérico Divide y Vencerás.

En el Código 2, el parámetro N realiza una doble función, la de representar el número de procesadores disponibles y la de representar el tamaño del problema. Además si se calcula el número de procesadores que intervienen en la resolución de las distintas subtareas no es necesario

utilizar N. Asumamos que un problema se considera trivial si su tamaño es uno. Consideremos una llamada al procedimiento DC con tamaño del problema dieciséis. El número de procesadores que finalmente trabajarán en la resolución de las distintas subtareas será ocho. Por lo tanto, sería suficiente inicializar diez procesadores antes de la llamada. En principio el conjunto total de procesadores formarían un único grupo (@=0) con nombres de procesador (#) en el rango 0..9. Cuando se ejecuta la sentencia fork[0..sqrt(N)-1] se deberían formar cuatro grupos de procesadores con nombres de grupo entre 0 y 3 cada uno de ellos conteniendo cuatro procesadores con nombres # entre 0 y 3. Sin embargo, el grupo @=2 solo contiene a dos procesadores con nombres 0 y 1, mientras que el grupo @=3 es vacío. De lo expuesto concluimos que si se pretende un algoritmo que utilice menos procesadores es necesario reescribir el algoritmo de manera que no coincidan el número de procesadores disponibles y el tamaño del problema.

En la segunda representación de DC (Código 3) que proponemos, se divide nuevamente el problema en $\sqrt{N}$ subproblemas de tamaño $\sqrt{N}$, pero ahora se puede asegurar que ninguno de los grupos de procesadores que se han de encargar de la resolución de los mismos será vacío, puesto que el número total de procesadores se distribuye cíclicamente.

```
/*    N denota el tamaño del problema    */
procedure DC( shared const N : integer; …);
…
if trivial(N) then conquer(…)
else
  fork [0..sqrt(N)-1]
  @ = # mod sqrt(N);
   # = # div sqrt(N);
     DC( sqrt(N), …) /* sqrt(N) denota ⌈√N⌉ */
   endfork
endif;
…
```

Código 3. Una Alternativa para el Algoritmo del Código 2.

2.1.2.3 La Sentencia de Asignación.

Asumamos que se ejecuta una sentencia de asignación de la forma:

$<expr>_1 := <expr>_2$

En primer lugar, todos los procesadores de *G* evalúan síncronamente la segunda expresión. Cada procesador de *G* evalúa el lado izquierdo de la asignación a una variable privada o compartida. Entonces los procesadores de *G* síncronamente asignan el valor del lado derecho a la variable descrita por el lado izquierdo.

El efecto de esta acción se define de la siguiente forma: Si la variable es privada, después de la asignación ésta contiene el valor escrito por el procesador. Si la variable es compartida, entonces según el régimen de resolución de conflictos de escritura elegido se determina el éxito o el fracaso de la asignación, y en caso de éxito el valor se le asigna a la variable.

Consideremos que cuatro procesadores numerados del 0 al 3 ejecutan la asignación*:* *x* := @ + # donde *x* es una variable de tipo entero. El valor de *x* después de la asignación depende de si *x* fue declarada como privada o como compartida. Podemos contemplar los siguientes casos:

- *x* es una variable privada. En este caso hay cuatro variables *x* distintas. Después de la asignación *x* contiene para cada procesador el valor de la expresión @ + #.
- Los cuatro procesadores forman un grupo hoja y la variable *x* es compartida con respecto a ese grupo hoja. En este caso los cuatro procesadores escriben en la misma variable y este conflicto se resolverá de acuerdo al régimen de resolución de conflictos de escritura.

- Los cuatro procesadores forman dos grupos hojas diferentes (por ejemplo, los procesadores 0 y 1 están en el primero y el 2 y 3 en el segundo). Cada grupo hoja tiene una instancia diferente de la variable *x*. Entonces los procesadores 0 y 1 escriben en la misma variable y lo mismo ocurre para los procesadores 2 y 3. El régimen de resolución de conflictos de escritura determinará el valor de las variables *x* (dos distintas) después de la asignación.

Se ha descrito la asignación de valores básicos. El lenguaje soporta asignaciones de valores estructurados (por ejemplo, a:= b; donde a y b son del mismo tipo array) que se lleva a cabo componente a componente síncronamente.

¿Qué sucede cuando varios procesadores acceden a la misma variable compartida síncronamente?. En este caso, el fracaso o el éxito de la operación y el efecto de la misma, se determina de acuerdo al régimen fijado inicialmente para resolver tales conflictos. En la mayoría de los algoritmos para el modelo PRAM se permiten operaciones de lectura concurrente (CREW). Sin embargo, FORK no se limita a este modelo.

Consideremos por ejemplo el régimen en el que las escrituras concurrentes están permitidas y el resultado se determina de acuerdo a los números de los procesadores implicados, esto es, el procesador con el menor número gana.

```
start [0..2*N-1]
   …
    shared var A: array[0..N-1] of integer;
    …
    fork [0..1]
    @ = # div N;
    # = # mod N;
      A[#] := result(A, @,#);
    endfork
    …
endstart
…
```

Código 4. Ejemplo FORK de Escrituras Concurrentes.

En el fragmento de programa FORK del Código 4, la variable compartida *A*, se declara para un grupo G que tiene asignados procesadores con números 0,...,$2 \times N - 1$. Para $n \in \{0,\ldots,N-1\}$ existen procesadores $p_{n,0}$ y $p_{n,1}$ con número de procesador *n* en el primero y el segundo de los grupos hoja, respectivamente que tratan de asignar un valor a *A[n]*. Este conflicto se resuelve de acuerdo con los números de procesadores de $p_{n,0}$ y $p_{n,1}$ relativos al grupo G, esto es , *n* y *N*+*n* respectivamente. De esta forma el resultado del procesador $p_{n,0}$ es almacenado en *A[n].* Nótese que este esquema fracasa si se introduce otra sentencia *start* dentro de la sentencia *fork* porque el número de procesador original no se puede determinar. En este caso la ejecución del programa falla.

2.1.2.4 Formación Implícita de Grupos: La Sentencia **if.**

Asumamos que se ejecuta una sentencia if de la forma:

---

if <*expr*> ***then*** <*Sents* >$_1$ ***else*** <*Sents* >$_2$ ***endif***

Entonces todos los procesadores de G en primer lugar evalúan síncronamente la expresión <*expr*>. Dependiendo del resultado de esta evaluación los procesadores ejecutan síncronamente las sentencias de la parte del *then* o del *else*, respectivamente. Puesto que procesadores diferentes pueden evaluar <*expr*> a valores diferentes no se puede asegurar que todos ellos continúen trabajando síncronamente. La jerarquía de grupo y los nuevos grupos síncronos maximales que ejecutarán las partes del *then* y del *else* (si está presente), respectivamente, se determinan de acuerdo a las constantes, variables y funciones de las que dependa la expresión <*expr*>. Precisamente, se dirá que <*expr*> depende de una variable *x* si *x* está presente en <*expr*> no como parámetro actual de una llamada a función. El caso de las constantes y funciones es análogo. Se han de tratar dos casos:

1.- La expresión no depende de ninguna variable, constante o función privada.

2.- La expresión depende de variables, constantes o funciones privadas.

En el primer caso, como la expresión no depende de datos privados, no se modifica la jerarquía de grupo H. Sólo puede cambiar la sincronización entre procesadores. Consideremos un procesador p de G. Se escoge como grupo síncrono maximal que contiene a p, el grupo maximal g_p de H que satisface las siguientes condiciones:

- g_p es un subgrupo de G (no necesariamente propio).
- p está contenido en g_p.
- La condición <*expr*> no depende de ninguna variable compartida o de otro dato compartido relativo a un subgrupo propio de g_p.

Bajo estas condiciones todos los procesadores en g_p evalúan <*expr*> al mismo valor. Entonces todos los procesadores escogen la misma ramificación de la sentencia if y así están en el mismo punto del programa. Nótese que un procesador fuera de g_p opera asíncronamente con el procesador p incluso si éste evalúa la expresión <*expr*> al mismo valor. Cuando un procesador de G ha finalizado la ejecución de <*Sents*> espera hasta que los otros procesadores de G finalicen sus secuencias de sentencias. Cuando todos los procesadores de G hayan alcanzado el final de la sentencia if, G se convierte nuevamente en síncrono maximal. Esto significa que incluso si dos procesadores de G trabajan asíncronamente dentro de la sentencia if, se convierten en síncronos nuevamente después de la ejecución de la sentencia if.

Consideremos que durante la ejecución de un programa se obtiene la jerarquía de grupo de la Figura 1 y todos los procesadores ejecutan síncronamente la sentencia

if x = 5 then S1 else S2 endif

donde una instancia de x es un variable compartida relativa a G_1, y otra instancia de x es una variable compartida relativa a G_2.

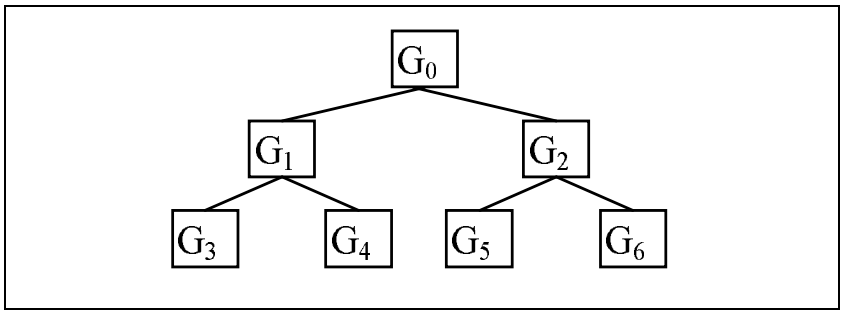


Figura 1. Una Jerarquía de Grupo.

Entonces, los procesadores del grupo G_3 trabajan síncronamente con los procesadores del grupo G_4 y lo mismo sucede para los grupos G_5 y G_6, respectivamente. Los procesadores del grupo G_4 trabajan asíncronamente con los procesadores del G_6 incluso si las dos instancias de la variable x contienen el mismo valor.

En el caso de que la expresión de una sentencia if dependa de variables, constantes o funciones privadas, no se puede estar seguro de que todos los procesadores dentro del mismo grupo hoja evalúen <*expr*> al mismo valor. La jerarquía de grupo H y la sincronización se modifican de la siguiente manera: cada grupo hoja de G genera dos nuevos grupos hoja; el primero contiene todos los procesadores del grupo hoja que evalúan <*expr*> a cierto, y el segundo contiene al resto. Todos los nuevos grupos hoja obtienen el número de grupo de su grupo padre. Los nuevos grupos hoja se convierten en síncronos maximales. Entonces, los procesadores dentro del mismo subgrupo, trabajan síncronamente, mientras que los procesadores de subgrupos diferentes trabajan asíncronamente. Nuevamente, cuando un procesador alcanza el final de una sentencia *if*, éste espera hasta que los otros procesadores alcancen este punto. Cuando todos los grupos hoja generados recientemente hayan terminado la ejecución de la sentencia *if*, esto es, cuando todos los procesadores hayan alcanzado el *endif*, los grupos hoja son eliminados. La jerarquía de grupo original H se restablece y G es de nuevo síncrono maximal.

Asumamos que para algún algoritmo los procesadores 1,…,N están organizados en forma de una árbol de altura *log(N).* En un instante *t*, un procesador debería ejecutar un procedimiento *op1(#)* si la altura del árbol es al menos *t*, y en otro caso el procesador debería ejecutar *op2(#).* El programa podría ser como el del Código 5.

```
shared var t : integer;
…
for t := 1 to log(N) do
  if height(#) <= t then op1(#)
    else op2(#)
  endif
enddo
…
```

Código 5. Ejemplo de una Sentencia if-then-else.

Para cada *t*, la condición puede ser evaluada a cierto o falso por diferentes procesadores. Además la ejecución de *op1(#)* y *op2(#)* puede introducir variables compartidas locales que son distintas incluso si tienen los mismos nombres. Además, cada sentencia if-then-else cuya condición dependa de variables privadas introduce implícitamente nuevos grupos hoja de procesadores, aquellos que evalúan su condición a cierto forman uno y los que la evalúan a falso, otra. Claramente dentro de cada grupo hoja todos los procesadores están en el mismo punto del programa. De esta forma pueden trabajar síncronamente. Tan pronto como los grupos hojas hayan finalizado las partes del *then* y el *else* respectivamente, se restablece el grupo hoja original y se procede con la ejecución síncrona de la siguiente sentencia. En el programa del ejemplo, la condición del bucle *for* sólo depende de la variable compartida *t*. Por lo que el grupo hoja actual no se subdivide en subgrupos. Sin embargo, la subdivisión ocurre en la línea siguiente. Los dos grupos, para el *then* y el *else* ejecutan sus acciones correspondientes en paralelo, cada grupo es internamente síncrono pero asíncrono con respecto a los procesadores de otros grupos. Al finalizar se restablece el grupo hoja original que síncronamente ejecuta la siguiente iteración del bucle y así sucesivamente.

2.1.2.5 La Sentencia **while.**

Una sentencia while tiene la siguiente sintaxis:

while <*expr*> ***do*** <*Sents* > ***enddo***

Su semántica viene determinada por la semántica de una sentencia *if-then-else* y la recursividad. La semántica de otra clase de bucles, esto es, bucles *for* o *repeat*, se pueden reducir a la del bucle *while*.

Obsérvese que en cada punto del programa el sistema de grupos y subgrupos que contienen a un procesador determinado forman una jerarquía. Correspondiéndose a esta jerarquía, las variables compartidas se pueden organizar en forma de árbol. Cada nodo se corresponde con un grupo de la jerarquía y contiene las variables compartidas relativas al grupo. Para un procesador de un grupo hoja todas aquellas variables relacionadas están situadas en el camino desde este grupo hoja a la raíz. Por supuesto a lo largo de este camino se han de mantener las reglas de ámbito. Para devolver resultados al final de un fork o para intercambiar datos entre diferentes grupos hoja es necesario tener al menos en algunos casos un acceso síncrono a los datos compartidos entre diferentes grupos de procesadores. Consideremos como ejemplo el Código 6.

```
…
shared var A: array[0..N-1] of integer;
shared var i:integer;
…
fork [0..N-1]
@ = …;
# = …;
   for i:=0 to N - 1 do
      A[(@+1) mod N] := result(i, @,A);
   enddo
endfork
…
```

Código 6. Ejemplo de Ambito y Jerarquía de Variable Compartidas.

En este ejemplo el array *A* se utiliza como un buzón de comunicación entre los grupos *1,…,N*. El índice *i* y los límites del bucle *for* son compartidos no sólo por los procesadores dentro de cada grupo hoja, sino también por todos los grupos que se generan en la sentencia *fork* . Si la condición del bucle depende sólo de variables compartidas por todos los grupos existentes, la semántica de FORK garantiza que el bucle se ejecutará síncronamente en esos grupos. Así los resultados calculados en la iteración *i* estarán disponibles para todos los grupos en la iteración *i+1*.

2.1.2.6 Las Llamadas a Procedimiento.

La sintaxis y la semántica de las llamadas a procedimiento son similares a las de Pascal. Hay algunas diferencias debido al hecho de que los procesadores pueden acceder a dos clases diferentes de objetos: los compartidos y los privados. Esto impone algunas restricciones sobre los parámetros actuales de una llamada a procedimiento. Sólo se ha de ser cuidadoso en los tipos de accesos de los parámetros formales y actuales:

- Si un parámetro formal es un PARÁMETRO POR REFERENCIA COMPARTIDO entonces el parámetro actual correspondiente tiene que ser una variable que no dependa de ningún dato privado, esto es, ar[#] no estaría permitida como un parámetro actual para un parámetro formal por referencia compartido, incluso si "ar" es un array compartido.
- Si un parámetro formal es un PARÁMETRO POR REFERENCIA PRIVADO entonces el correspondiente parámetro actual puede ser una variable privada o compartida.
- Si un parámetro formal es un PARÁMETRO CONSTANTE COMPARTIDO O PRIVADO entonces el correspondiente parámetro actual puede ser un valor privado o compartido.

El caso en el que un parámetro formal POR REFERENCIA COMPARTIDO sea sustituido en una llamada con una variable privada está excluido explícitamente puesto que esto permitiría a algunos procesadores modificar el contenido de esta variable privada, que puede pertenecer a un

procesador diferente. En contraste, en el caso de un parámetro formal CONSTANTE COMPARTIDO el valor del parámetro formal durante la ejecución del procedimiento está determinado por el régimen para resolver los conflictos de escritura. Esto se hace de la misma forma que cuando se determina el valor de una variable compartida después de la asignación de un valor privado.

Asumamos que el grupo síncrono maximal G ejecuta la llamada a procedimiento:

$$<identificador>(<expr>_1,\ldots,<expr>_n)$$

Primero se evalúan síncronamente los parámetros actuales $<expr>_1,\ldots,<expr>_n$ y se hacen corresponder con lo parámetros formales síncronamente de izquierda a derecha. Entonces G ejecuta síncronamente el bloque del procedimiento, esto es, las declaraciones y las sentencias del procedimiento.

2.1.2.7 Expresiones.

La sintaxis y la semántica de las expresiones es la misma que la de Pascal. Hay dos nuevas constantes predefinidas: el *número de procesador* # y el *número de grupo* @ que son privadas de tipo entero. Cada procesador de un grupo síncrono maximal G evalúa la expresión y devuelve un valor. El valor devuelto por una función compartida se determina de acuerdo al régimen de resolución de los conflictos de escritura de forma separada por cada grupo hoja de G y es tratado como un dato compartido de este grupo hoja. Nótese que la evaluación de expresiones puede provocar conflictos de lectura, que se resolverán de acuerdo con el régimen de resolución elegido. Consideremos el ejemplo del Código 7.

```
...
shared var a : array [1..10] of integer;
...
... := a[3] + 3
...
```

Código 7. Evaluación de una Expresión.

Determinar las direcciones correspondientes a la subexpresión a[3] no provoca conflictos de lectura, mientras que determinar el valor de esa variable puede provocar conflictos de lectura.

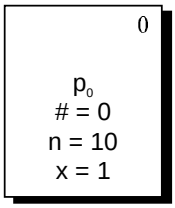
2.2 Un Ejemplo de Programa FORK.

En esta sección se presenta un ejemplo sencillo de un programa FORK (Código 8) y se describe la semántica de una manera informal. Se muestra cómo las diferentes construcciones de FORK cambian la jerarquía y las jerarquías síncronas maximales que ejecutan un programa.

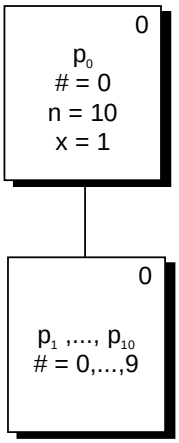
```
(1) begin
(2)    shared const n = 10;
(3)    shared var x : integer;
(4)    x := 1;
(5)    start [0..n - 1]
(6)       fork [0..1]
(7)       @ = # mod 2;
(8)       # = # div 2;
(9)       begin
(10)          shared var y : integer;
(11)          y := x + @;
(12)          if y = 1 then
(13)             y := 2
(14)          else
(15)             y := 3
(16)          endif;
(17)          if # < 2 then
(18)             x := x + y
(19)          else
(20)             y := y + #
(21)          endif;
(22)       end
(23)    endfork
(24)  endstart
(25) end
```

Código 8. Un Ejemplo Sencillo de un Programa FORK.

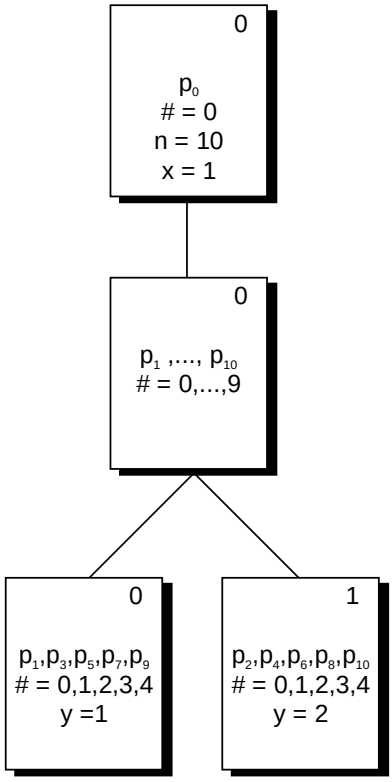
En la línea 1, sólo existe un grupo que contiene un único procesador p_0 con número de procesador 0. El número de grupo de este grupo es 0. La jerarquía de grupo consta de este grupo y es síncrona maximal. Cuando la ejecución del programa alcanza la línea 5, se tiene la siguiente jerarquía de grupo:



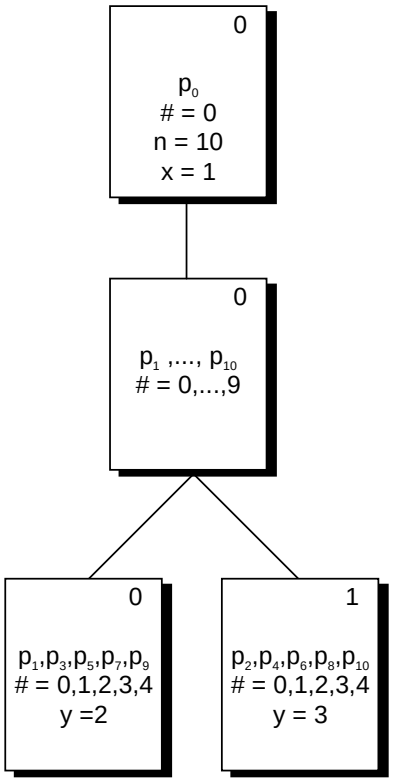
En la línea 5, la jerarquía se expande en un nuevo grupo que consta de 10 procesadores p_1, p_2, ..., p_{10} con números 0,...,9. Estos procesadores trabajan síncronamente. El procesador original p_0 detiene su trabajo durante la ejecución de la sentencia *start*. La nueva jerarquía queda de la siguiente forma:



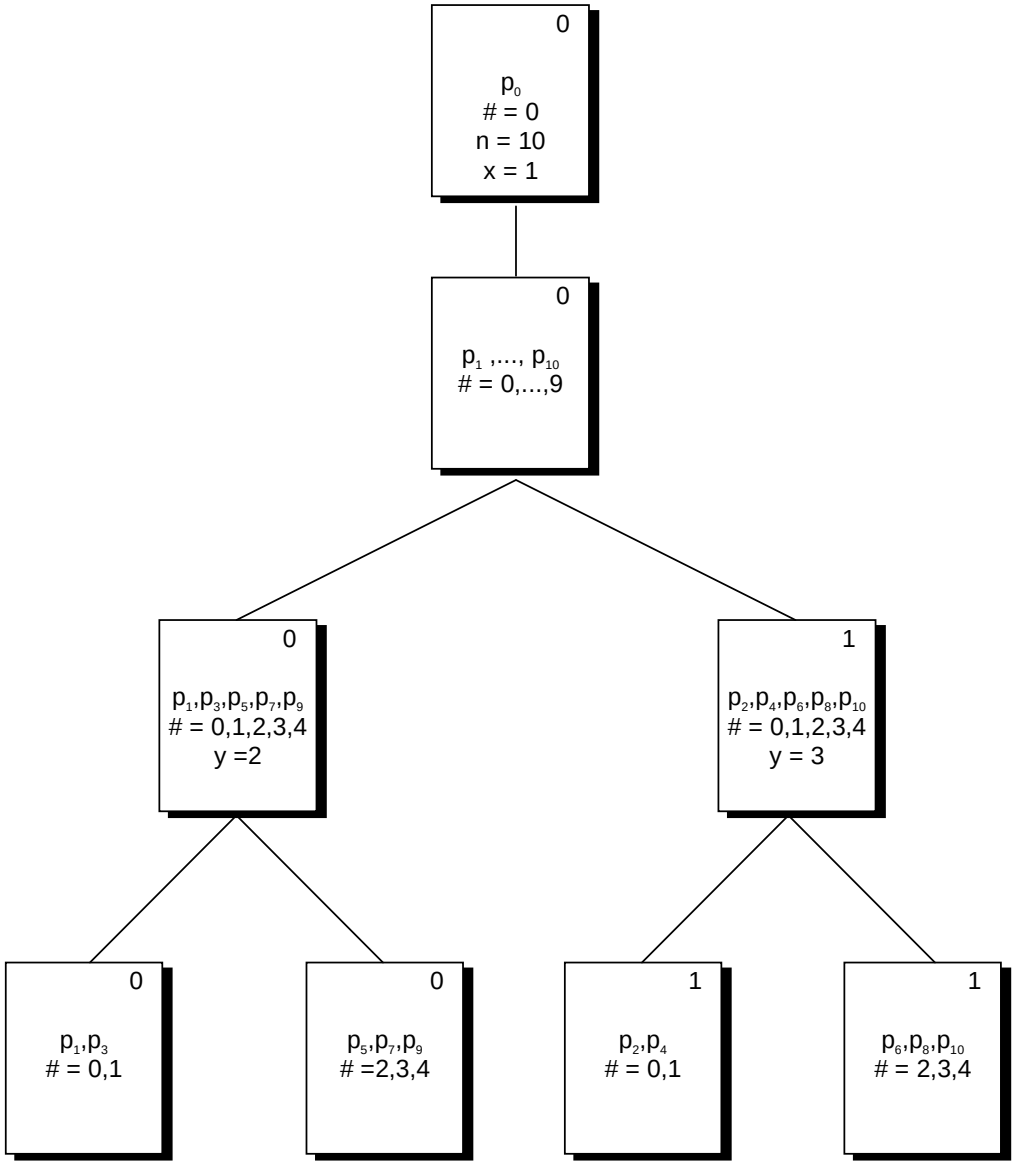
Esta jerarquía es síncrona maximal ejecutando las sentencias dentro del *start*. En las siguientes figuras se muestra el conjunto de procesadores que pertenecen a cada grupo. Esto no significa que existan varias instancias de un procesador específico trabajando en varios grupos. Pretendemos expresar que durante una computación el procesador estaba activo y vuelve a estar activo nuevamente en otro grupo. Los procesadores activos son siempre miembros de grupos hoja. En las líneas 6-11 el nuevo subgrupo de procesadores se divide en dos subgrupos cada uno con su variable compartida “y” que se inicializa a 1 ó a 2 respectivamente. Entonces la jerarquía cambia a:



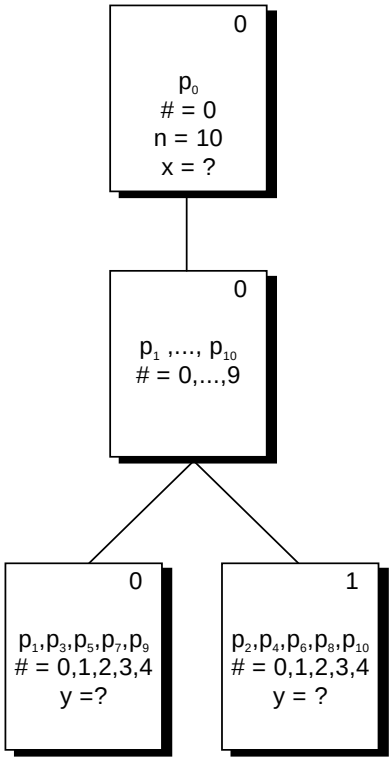
Esta jerarquía también es síncrona maximal. En la línea 12 los procesadores p_0 ,...,p_9 realizan una comprobación sobre una variable que es compartida con respecto a un subgrupo minimal (esto es, un grupo hoja) de la jerarquía síncrona maximal. Esto implica que las sentencias y:= 2 e y := 3 son ejecutadas asíncronamente y en paralelo por las subjerarquías, que están determinadas por el grupo minimal. Por lo tanto, los dos grupos minimales forman la jerarquía síncrona maximal durante la ejecución de las ramificaciones de la sentencia if-then-else. Cuando ambas jerarquías síncronas maximales han terminado la ejecución de su ramificación, han cambiado el valor de “y” a 2 ó 3, y la jerarquía síncrona maximal estará representada por:



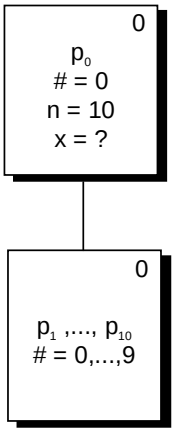
Esta jerarquía síncrona maximal comienza la ejecución del la sentencia if-then-else de la línea 17. Puesto que la condición depende de un dato privado (el número del procesador, #) , cada grupo minimal de la jerarquía síncrona maximal se divide en dos subgrupos: uno contiene los procesadores que evalúan la condición a cierta y el otro contiene a los procesadores que la evalúan a falso. Entonces para ejecutar las ramificaciones la jerarquía cambia a:



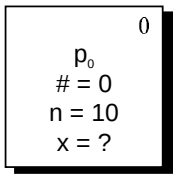
y los grupos hoja de esta jerarquía determinan una jerarquía síncrona maximal. Esta última ejecuta la ramificación que le corresponde. Cuando todas las jerarquías síncronas maximales terminan la ejecución de su ramificación, se obtiene de nuevo la única jerarquía de la forma:



que alcanza el *endfork* de la línea 23. Este *endfork* cambia la jerarquía a:



y el *endstart* de la línea 24 cambia la jerarquía a:



En este punto el procesador p_0 continúa trabajando.

2.3 Consideraciones acerca de la Implementación.

En esta sección se esbozan algunas ideas que muestran que los programas FORK no sólo pueden ser traducidos a código PRAM semánticamente equivalente, sino que el código se ejecuta eficientemente. Estas consideraciones se suponen útiles tanto para los teóricos como para los diseñadores de compiladores, que puedan tener diferentes realizaciones de la PRAM disponibles, posiblemente sin sistemas operativos potentes para la manipulación de memoria y asignación de procesadores. La idea original de los autores para compilar FORK era extender la implementación usual de llamadas a procedimientos recursivos basándose en una pila , esto es, la P-máquina con un correspondiente régimen para las estructuras compartidas, para los mecanismos de sincronización y para la manipulación de los números de grupo y de procesador. Se discutirá sobre las siguientes ideas:

- Creación de nuevos subgrupos
- Sincronizaciones
- Inicializaciones de nuevos procesadores con *start*.

2.3.1 Creación de Subgrupos.

Las variables que son compartidas con respecto a algún grupo se han de colocar en alguna porción de la memoria compartida de la PRAM, la cual estará reservada para ese grupo. Entonces, el punto crucial al crear nuevos subgrupos es la cuestión de cómo los subgrupos obtienen porciones distintas de la memoria compartida. Existen dos formas (al menos) de conseguir esto con una sobrecarga computacional aceptable:

1. Mediante direccionamiento aritmético
2. Teniendo en cuenta que en la práctica la memoria compartida disponible es siempre finita, subdividiendo en partes iguales el

espacio que permanece libre entre las nuevas creaciones de grupos.

El primer método se corresponde con un esquema de direccionamiento donde el almacenamiento restante se ve como una matriz de dos dimensiones. Sus filas están indexadas por los números de grupo, mientras que los índices de las columnas proporcionan las direcciones de una celda de almacenamiento relativa a un grupo dado. Para el segundo método el papel de las filas y las columnas simplemente se intercambia. En ambos casos la división en subgrupos se puede ejecutar en tiempo constante. Este esquema de asignación de memoria está bien adaptado a la jerarquía de grupos con requerimientos de espacio equitativos. Se puede alcanzar un desperdicio exponencial de espacio en otros casos. Consideremos el siguiente bucle while, cuya condición depende de una variable privada:

```
(1)  while cond(#) do
(2)      work(#)
(3)  enddo
```

Cuando el grupo de procesadores trabaja síncronamente ejecutando la línea 1, este es subdividido en dos grupos, uno consta de los procesadores que no satisfacen *cond(#)* , y el otro consta de los restantes procesadores. Así, el primer grupo no necesita espacio de memoria compartida (de todos modos quizás alguna cantidad constante por razones de organización), una subdivisión sencilla en dos porciones iguales debería innecesariamente dividir en la mitad el espacio disponible para el segundo grupo para ejecutar el *work(#)* de la línea 2. Sin embargo, hay una optimización inmediata del esquema de distribución del almacenamiento propuesto anteriormente: asignar una cantidad constante y fija (conocida en tiempo de compilación) de espacio a los grupos que no trabajan puesto que no necesitan nuevo espacio de memoria compartida y subdividir el espacio restante de forma equivalente entre los otros subgrupos.

Esta optimización se puede realizar de manera automática en bucles como los del ejemplo considerado y también en una una sentencia if, o por las sentencias if -then-else en las que una alternativa incluya un bloque con una parte de declaraciones vacía.

2.3.2 Sincronizaciones.

En orden de restablecer un grupo g, el sistema de tiempo de ejecución tiene que determinar cuando todos los subgrupos de grupo g han terminado. Este es el problema de detección de terminación para los subgrupos. Si todos los procesadores corren síncronamente, no es necesaria ninguna sincronización explícita. En el caso general, donde los subgrupos del grupo g corren asíncronamente, existen las siguientes posibilidades para implementar la detección de terminación:

1. Usar un soporte hardware especial tal que en una unidad de tiempo una operación de fetch&add permita a los procesadores dentro de un grupo simultáneamente añadir un entero a una variable compartida.
2. Análisis estático (asistido por anotaciones del usuario); la mayoría de los algoritmos PRAM publicados en la literatura tienen una estructura tan simple y regular que los tiempos de ejecución relativos a secuencias se pueden determinar de esta manera.
3. Usar un algoritmo de detección de terminación en tiempo de ejecución. Esto último siempre es posible aunque ineficiente. Los programas complejos que engañen al analizador estático pagarán con una pérdida de eficiencia extra.

2.3.3 Inicialización de Procesadores.

El siguiente método es análogo al del esquema de distribución del almacenamiento y trabaja sólo para máquinas Concurrent-Read con un número finito de procesadores. Antes de ejecutar el programa, todos los procesadores son inicializados con número de procesador # = 0. Si en una

sentencia *start* posterior del programa se inicializan menos procesadores de los que están físicamente disponibles en el grupo hoja ejecutando esa sentencia, entonces varios procesadores físicos pueden permanecer "idénticos", esto es, recibir el mismo nuevo número de procesador. Estos procesadores idénticos eligen un líder. Todos ellos ejecutan el programa pero sólo al líder se le permite realizar operaciones de escritura en variables compartidas. Consideremos el ejemplo del Código 9. Asumimos que se tienen 512 procesadores físicos.

```
(1)   start [0..127]
(2)     if # < 64 then
(3)       start [0..212]
(4)         compute(212)
(5)       endstart
(6)     endif
(7)   endstart
```

Código 9. Inicialización de Nuevos Procesadores.

Antes de la línea 1, todos los 512 procesadores físicos son inicializados. Después de la línea 1, hay siempre cuatro procesadores con el mismo número de procesador #. Habiendo ejecutado la condición de la línea (2) todos los procesadores cuyo número de procesador sea menor que 64 entran en la parte del then: esto es 256. Todos ellos están disponibles para la ejecución de la sentencia *start* de la línea 3, donde reciben los nuevos números 0,...,212. Dos procesadores físicos se asignan a cada uno de los primeros 43 procesadores lógicos, mientras que un procesador físico se asigna a cada uno de los 170 procesadores lógicos restantes. Las identidades de los procesadores originales se almacenan en la pila privada del sistema. Cuando se alcanza el *endstart* de la línea 5, los procesadores restablecen su números de procesador anteriores. Si en el grupo actual se inicializan más procesadores de los que hay disponibles físicamente, entonces cada procesador dentro del grupo tiene que simular a un subconjunto apropiado de los nuevos procesadores inicializados. En ambos casos *start* se puede ejecutar en tiempo constante por cada grupo hoja que conste de un intervalo contiguo de procesadores: este es el caso, por

ejemplo, para los *starts* que aparecen en la secuencia de sentencias en el nivel más alto del bloque.

Se hace necesario entonces, un estilo de programación donde los procesadores lógicos necesarios para la ejecución de un programa sean inicializados al comienzo, esto es, antes de dividir el grupo inicial en subgrupos, o bien, sean inicializados de una manera equilibrada por subgrupos contiguos.

Para maximizar la explotación de los recursos de la arquitectura PRAM dada, un programador podría escribir programas en los cuales use diferentes algoritmos para números de procesadores físicos disponibles diferentes. Entonces, se debería proporcionar una constante de sistema (compartida) de tipo entero, cuyo valor sea el número de procesadores físicos disponibles en la PRAM dada. Esto permitiría a los programas adaptarse al hardware subyacente.

2.4 ¿Por qué FORK no Introduce una Sentencia Paralela Clásica?

Los algoritmos para el modelo PRAM se suelen describir en un estilo general que no necesita una definición formal para los lectores con experiencia en programación secuencial. La admisión de paralelismo en la mayoría de los casos se hace a través de una sentencia paralela del tipo:

for all $x \in X$ ***in parallel do*** *instrucción(x)*

Donde x es un elemento del conjunto X y la ejecución de la sentencia consiste en:

(a) Asignar un procesador a cada elemento $x \in X$

(b) Ejecutar, en paralelo y por el procesador asignado, todas aquellas operaciones especificadas por *sentencia*.

Cuando todos los procesadores involucrados completan sus cálculos individuales se detiene la ejecución.

De esta definición surgen dos preguntas: ¿cómo se asignan nombres a los procesadores? y ¿cómo se activan los procesadores?.

La asignación de procesadores requiere que los elementos de *X* sean enteros distintos o se puedan codificar mediante enteros distintos. Entonces, con cada *x* se asocia el procesador $P_{code(x)}$ donde *code(x)* es el entero correspondiente a *x*. Se asume que un procesador distinguido, el procesador control, dirige al procesador *i* a trabajar como $x=code^{-1}(i)$ en tiempo constante.

En la activación de procesadores, se asume que hay un umbral de tiempo inicial de *log(p)* para activar *p* procesadores al comienzo de cualquier computación. Donde *p* es el número máximo de procesadores requeridos durante la computación. Esto es consistente con que se active un procesador al principio de la ejecución y posteriormente en cada paso de tiempo el número de procesadores activos se pueda doblar haciendo que cada procesador activo active a otro.

Esta definición de sentencia paralela [Gib88] asume implícitamente un costo logarítmico en la activación de procesadores cada vez que es utilizada. En contraposición a esta política se encuentran en la literatura otro tipo de definiciones [Qui94] en las que dicho costo se considera de forma separada. Puesto que un algoritmo PRAM comienza con sólo un procesador activo, los algoritmos PRAM tienen dos fases: en la primera fase se activa el número suficiente de procesadores y en la segunda esos procesadores activos realizan la computación en paralelo. Dado un único procesador activo, para activar *p* procesadores, son necesarios y suficientes $\lceil \log p \rceil$ pasos de activación, puesto que el número de procesadores activos puede doblarse en cada paso.

En la presentación de los algoritmos PRAM se suele usar la meta-sentencia:

spawn (<Nombres de procesador>)

para denotar el tiempo logarítmico de generación de procesadores por parte de un único procesador.

La sentencia:

for all <*lista de procesadores*> ***do*** <*lista de sentencias*> ***endfor***

denota un segmento de código a ser ejecutado en paralelo por todos los procesadores especificados, asumiendo que estos han sido activados previamente.

Según los diseñadores de FORK la no inclusión de una sentencia paralela del tipo descrito está motivada por el hecho de que generalmente se utilizaría con el mismo sentido de la sentencia *start*. Sin embargo, se presenta una diferencia con las sentencias paralelas anidadas. Consideremos el siguiente fragmento de programa:

```
for all  i ∈ 1..N in parallel do
  for all  j ∈ 1..M in parallel do
    A[i,j] := 0;          /* A es una matriz compartida */
```

Si se utiliza una semántica similar a la de la sentencia *start* de FORK, la segunda sentencia paralela sobrescribe a la primera, lo que significa que solamente *M* procesadores ejecutan la asignación *A[i,j]:=0*. Además el valor de la variable *i* no está definido.

Evidentemente este no es el significado deseado. Se quiere que las dos sentencias paralelas anidadas inicialicen N×*M* procesadores para indexar los pares *(i,j)* . De alguna forma la sentencia *start* se corresponde con la meta-sentencia *spawn* y la sentencia *fork* con la sentencia paralela, pero la correspondencia no es directa.

Una sentencia paralela de la forma:

for all $i \in < expr >_1 .. < expr >_2$ ***in parallel do***
<*Sentencia*>

donde las expresiones $< expr >_1$ y $< expr >_2$ y la *sentencia* <*Sentencia* > no usan ningún dato privado, se puede simular como se muestra en el Código 10.

```
begin
   /* Se declaran dos constantes auxiliares para evitar la doble */
   /* evaluación de las expresiones <expr>1  y <expr>2  */
   shared const a1 =  <expr>1;
   shared const a2 =  <expr>2;
   /* Se inicializan a2 - a1 + 1 nuevos procesadores…*/
   start[a1..a2]
      /* … y se distribuyen entre a2 - a1 + 1 nuevos grupos */
      fork[a1..a2]
         @ := #;
         # := 0;
         /* cada grupo hoja crea una variable i nueva y la inicializa con el número de grupo*/
         begin
            shared var i : integer;
            i:= @;
            <sentencia>
         end
      endfork
   endstart
end /* Final de la simulación de una sentencia paralela*/
```

Código 10. Simulación en FORK de una Sentencia forall Clásica.

Por lo tanto, para evitar redundancias se decidió no incluir una sentencia paralela en FORK, aunque con ello se pierde un poco de elegancia a la hora de presentar ciertos algoritmos mientras que se gana en la presentación de otros. Siguiendo el esquema presentado, el Código 11 muestra la inicialización de la matriz compartida *A.*

```
begin  /* for all  i ∈ 1..N in parallel do */
   shared const a1 =  1;
   shared const a2 =  N;
   start[a1..a2]
      fork[a1..a2]
         @ := #;
         # := 0;
         begin
            shared var i : integer;
            i:= @;
            begin  /* for all  j ∈ 1..M in parallel do */
               shared const b1 =  1;
               shared const b2 =  M;
               start[b1..b2]
                  fork[b1..b2]
                  @ := #;
                  # := 0;
                     begin
                        shared var j : integer;
                           j:= @;
                           A[i,j] := 0;
                     end
                  endfork
               endstart
            end /* for all  j ∈ 1..M in parallel do */
         end
      endfork
   endstart
end /* for all  i ∈ 1..N in parallel do */
```

Código 11. Programa FORK para Inicializar una Matriz "A" Compartida.

3. Fork95: Un Lenguaje Basado en FORK.

El diseño original de FORK presentado en la sección anterior es teórico. Para permitir una definición semántica rigurosa de las sentencias paralelas *fork* y *start* los autores insisten en una organización rígidamente modular (sin saltos) y excluyen cualquier otra estructura de datos que no sean arrays o registros. Aunque los autores son bastante restrictivos en estos aspectos, luego permiten la inicialización anidada de procesadores o que el número de procesadores inicializados pueda exceder el número de procesadores físicos existentes. Las modificaciones en el diseño que se presentan en [Kes94, Kes95a, Kes95b] pretenden eliminar estas deficiencias: El nuevo diseño restringe el ámbito de aplicación de la sentencia *start* y restringe la cantidad de programa para la que el lenguaje garantiza una ejecución síncrona. Según la opinión de los autores de la

implementación, la pérdida de expresividad, que califican de pequeña, se ve compensada con la obtención de eficiencia y una enorme reducción de la sobrecarga en tiempo de ejecución. Sin embargo, creemos que la relajación en las síncronizaciones entre grupos, al ejecutar una sentencia *fork*, hace más difícil la expresión de algunos algoritmos PRAM.

El dialecto de FORK que proponen en [Kes94] se ha convertido más o menos en un superconjunto de C. Para alcanzar este objetivo, sus autores, extendieron la sintaxis del ANSI-C [Ans89], por lo cual para las partes secuenciales se ha de adoptar la filosofía C. Esto también tiene un impacto importante sobre las sentencias *start* y *fork*. A continuación se describen las características básicas de *fork95*.

3.1 Características que diferencian a fork95 de FORK.

3.1.1 La sentencia *start*.

Los programas escritos en *fork95* son ejecutados por "*threads"* que los autores denominan procesadores puesto que son asignados estáticamente a procesadores físicos de la arquitectura objeto. Todos los procesadores de la PRAM sobre los que el usuario ha cargado el programa se inicializan en paralelo. Después de esto, sólo un procesador permace activo y comienza la ejecución del programa mediante una llamada a la función main().

Para activar procesadores se utiliza la sentencia *start*. Ejecutando:

---

start *(e) <Sent>*

---

el único procesador activo en primer lugar evalúa la expresión *e* (necesariamente de tipo entero) y obtiene un valor *v*. Entonces se activan *v* procesadores cada uno con un único *Identificador* (absoluto) *de Procesador* denominado `__PROC_NR__` numerado sucesivamente desde 0 hasta v-1.

Estos procesadores ejecutan síncronamente la sentencia <*Sent*>. Cuando finalizan se desactivan y sólo continua activo el procesador 0.

Si el valor *v* excede el número de procesadores físicos disponibles, tiene lugar un error en tiempo de ejecución. Este aspecto constituye una diferencia de la definición original, así como también el que esté prohibido el uso de sentencias *start* anidadas. Además el parámetro de *start* se hace opcional. Si se ejecuta ***start*** <*Sent* > todos los procesadores disponibles por hardware son inicializados. Para explotar el sincronismo que provee el hardware, los procesadores que se inicialicen han de trabajar en modo fuertemente síncrono ("*lockstep"*).

3.1.2 Variables Privadas y Compartidas.

De acuerdo con el diseño inicial, se tienen dos tipos de almacenamiento, denominados privado (identificado por la palabra reservada **pr**) y compartido (identificado por **sh**). Se asume que por defecto el almacenamiento privado. Consideremos por ejemplo la siguiente lista de variables compartidas :

sh int i**,** a**[**20**],***p**;**

se reserva espacio en la memoria compartida para i, a[0],..., a[19] y p. La declaración de p define un dato compartido de tipo puntero a un entero (puede apuntar a un entero compartido o a uno privado).

Los datos privados de un procesador no pueden ser accedidos directamente por otros procesadores. Para permitir referencias entre unos y otros cada procesador posee una variable privada que lo distingue: *$*, su *número de procesador*. Esta variable se inicializa con `__PROC_NR__`. Durante la ejecución de un programa, *$* almacena el *Identificador* (relativo) *de procesador* actual. También existe una variable especial compartida que almacena el *Identificador del grupo* actual, @. A dicha variable la denominaremos *número de grupo*. Los valores de estas variables se guardan y se reestablecen de manera automática al realizar las operaciones de formación de grupos. Sin embargo, el usuario es el

responsable de asignarle valores razonables (por ejemplo, en la sentencia *fork*).

Una expresión es privada si no se garantiza la evaluación al mismo valor por parte de todos los procesadores. En general, una expresión se considera privada si contiene al menos una subexpresion privada (por ejemplo, una variable).

3.1.3 La sentencia *fork*.

Los procesadores se organizarán en grupos. A un grupo de procesadores se le asigna una tarea determinada y los miembros de ese grupo trabajan conjunta y síncronamente en el cálculo de la solución. Los grupos pueden subdividirse en subgrupos que se corresponden con una posible subdivisión jerárquica de la tarea inicial. De aquí se sigue que en algún punto de la ejecución del programa, todos los grupos existentes están organizados en una jerarquía de grupo. Sólo los grupos hoja de una jerarquía de grupo están activos. Los subgrupos de un grupo se pueden distinguir por su número de grupo. El número de grupo de un grupo hoja de procesadores puede ser accedido por un procesador *p* miembro de ese grupo a través de la variable compartida @. Sólo se garantiza la ejecución síncrona para los procesadores de los grupos hoja, lo que significa que estarán siempre en el mismo punto del programa y ejecutarán el siguiente paso de computación simultáneamente en paralelo (nótese que se usa un régimen débil de Ejecución Síncrona de Programa, en la definición original, las jerarquías mayores también podían ejecutarse síncronamente).

Cada grupo puede tener sus propios datos compartidos, los cuales sólo pueden ser accedidos por sus miembros. Si un grupo hoja *g* ejecuta una definición de una variable privada *x* entonces cada procesador *p* de G crea una instancia distinta de *x* la cual es sólo accesible por *p*. En contraposición, si G ejecuta una definición de una variable compartida *x* entonces sólo se crea una instancia de la variable y puede ser accedida por todos los procesadores de G.

Inicialmente, la jerarquía de grupo H consta de un único grupo con @=0 que contiene a todos los procesadores que han sido inicializados. Todos los procesadores corren síncronamente, esto es, ejecutan el mismo programa en modo fuertemente síncrono ("*lockstep*"). Para crear nuevos subgrupos se usa la sentencia fork cuya sintaxis es la siguiente:

---

fork (e_1 ***;*** e_2 ***;*** e_3 ***)*** <*Sent*>

---

donde cada e_i es una expresión de tipo entero, y e_1 no depende de datos privados.

Asumamos que un grupo hoja *G* ejecuta esta sentencia *fork*. Entonces, primero cada procesador *p* de *G* calcula los valores $v_{p,1}$, , $v_{p,2}$ y $v_{p,3}$ de las expresiones e_1 , e_2 y e_3 respectivamente. Puesto que e_1 no depende de datos privados, todos los valores $v_{p,1}$ son iguales al mismo valor v_1 . Este valor determina el rango [0.. v_1 -1] del conjunto de números de grupo de los nuevos subgrupos de G que han sido creados. El valor $v_{p,2}$ devuelve el número de grupo del grupo hoja al que el procesador *p* pertenece. Mientras que el valor $v_{p,3}$ proporciona el nuevo número de procesador de *p* dentro de su nuevo grupo hoja. Teniendo entonces actualizados los valores de $ y @ los procesadores empiezan a ejecutar las sentencias <*Sent*>. No se asume que cada nuevo grupo sea escogido por algún procesador, por lo tanto algunos de los nuevos grupos hoja pueden ser vacíos. Una vez finalizada la ejecución de <*Sent*>, los grupos hojas se eliminan de la jerarquía de grupo y todos los procesadores continúan la ejecución dentro del grupo G. En *fork95* no se garantiza la ejecución síncrona entre los diferentes grupos hoja recién creados, por lo que como en *G* se debe garantizar una ejecución síncrona, es necesaria una sincronización explícita al final de <Sent>. Si no se especifican parámetros en una sentencia fork (***fork*** <*Sent*>) se asume que se crea un único grupo de procesadores con @=0 y $=0.

3.1.4 Las sentencias condicionales.

Consideremos un grupo hoja *G* ejecutando una sentencia:

if (e) S_1 ***else*** S_2

Si la expresión *e* sólo depende de datos compartidos, todos los procesadores calculan idénticos valores para *e* y entonces seleccionan la misma ramificación. La situación sin embargo es totalmente diferente si *e* depende también de datos privados. Entonces algunos de los procesadores de G pueden evaluar *e* a falso y otros a cierto. Esto implicaría que los procesadores de G pueden tomar caminos diferentes en el programa al mismo tiempo, es decir, se convierten en asíncronos. Esto se evita dividiendo G en dos subgrupos G_0 y G_1 en tiempo de ejecución donde G_0 contiene a todos los procesadores de G que evaluaron *e* a 0 y G_1 contiene a todos aquellos que evaluaron *e* a un valor distinto de 0. Los nuevos subgrupos G_i heredan el número del grupo de G. Son asíncronos y empiezan a ejecutar S_i. Cuando la ejecución de ambos S_1 y S_2 ha terminado, los grupos hoja G_i se eliminan y G continúa la ejecución síncrona del programa.

La sentencia *switch* de la forma:

switch (e) <*Sent*>

se trata de forma análoga a la sentencia *if*, en la que (en caso de que el valor de *e* dependa de datos privados) se crean tantos subgrupos como alternativas haya en el cuerpo de <*Sent*> .

3.1.5 Conflictos de escritura.

Existen muchas posibilidades acerca de lo que se puede hacer si varios procesadores acceden simultáneamente para escribir en la misma

instancia de una variable *x*. Si varios procesadores intentan escribir en la misma posición de memoria compartida en el mismo ciclo de reloj, el procesador con menor valor de `__PROC_NR__` ganará y escribirá su valor , esto es, se ha elegido como método de resolución de conflictos una "*Priority-CRCW-PRAM"*. El programador de *fork95* tiene la posibilidad de cambiar el valor de `__PROC_NR__` durante la ejecución del programa y esto, dentro de ciertos límites, influye sobre el método de resolución de un conflicto de escritura. Esta elección elimina una fuente de no determinismo en el lenguaje, como ocurriría en el caso de elegir por simplicidad del convenio *"Arbitrary-CRCW-PRAM"*. Así pues, hay sólo una fuente de no determinismo en el lenguaje que surge de los diferentes entrelazados de las partes asíncronas de un programa.

3.2 Aportaciones del Lenguaje fork95.

Una vez enumeradas las principales diferencias con respecto al diseño original, para realizar un estudio completo de *fork95* es necesario destacar las nuevas aportaciones, las cuales han surgido principalmente del hecho de tomar como lenguaje secuencial base al ANSI-C. En lo que sigue se realiza un estudio de las características avanzadas de *fork95*.

3.2.1 Granjas.

La formación implícita de subgrupos en las sentencias condicionales que dependen de variables privadas es bastante costosa. Este costo se puede evitar por medio de una nueva sentencia:

---

farm <*Sent*>

Esta sentencia garantiza la ejecución asíncrona del programa durante la ejecución de <*Sent*>. Existe un punto de sincronización entre los grupos hoja cuando finaliza la ejecución de <*Sent*>.

La sentencia *farm* es útil para agrupar porciones del programa donde los procesadores sólo realizan computaciones locales. Sin embargo, la omisión de la formación de subgrupos implícitos implica que se debe evitar la asignación de variables compartidas dentro de <*Sent*>. Especialmente, las funciones que se llamen deben tener sólo parámetros privados y devolver un valor privado (si hay alguno).

Se introducen los calificadores de tipo *sync* y *async* para distinguir entre las funciones ordinarias y aquellas que se pueden llamar dentro de una sentencia *farm*. Una función declarada de la forma:

```
sync int fun();
```

se puede invocar desde fuera de una sentencia *farm* pero no desde dentro. De la misma forma, una función:

```
async int fun();
```

sólo se puede usar dentro de una sentencia *farm*. Las funciones asíncronas no deberían contener ninguno de los nuevos constructos de *fork95* o llamadas a funciones calificadas como síncronas. De esta forma los programas C ordinarios pueden ejecutarse dentro de una sentencia *farm* sin cambios sintácticos. Por defecto las funciones son *async*.

3.2.2 Punteros y Heaps.

La innovación más importante de *fork95* con respecto a su precursor FORK, es que *fork95* soporta punteros. En la implementación de la SB-PRAM no existen áreas de almacenamiento privadas reales. Los diseñadores del hardware de la SB-PRAM [Abo90, Kel94] afirman que una gran cantidad de memoria compartida de fácil acceso es mucho mas flexible que usar una pequeña memoria compartida junto con una memoria privada para cada procesador. Para esta arquitectura el concepto del lenguaje de variables privadas se puede implementar mediante áreas dentro de una única memoria compartida que pertenecen a los procesadores para su uso privado. Es lícito que una variable puntero compartida pueda apuntar a

algún dato privado y viceversa, la cuestión de si es o no un buen estilo de programación se deja al programador. En cualquier caso, la prevención por parte del compilador de cualquiera de estos casos es mucho más complicada que su admisión.

Como es usual en C, existe un operador de dirección & (devuelve el L-value de una expresión) y el de indirección *. En contraste con C secuencial, sin embargo, hay dos clases de heap en los que pueden residir los datos creados dinámicamente, denominados *heap privado* (situado en la porción de memoria compartida perteneciente a cada procesador) y *heap compartido*. En consecuencia existen también dos rutinas de librería, denominadas *malloc* y *shalloc* . Ambas operan de la misma forma que el *malloc* de C, con la única excepción de que *malloc(n)* ejecutada por el procesador *p*, asigna *n* celdas en el *heap privado* de *p*, mientras que *shalloc(n)* ejecutada síncronamente por todos los procesadores del mismo grupo hoja G, asigna *n* celdas en la porción de memoria compartida asignada a G. Ambas devuelven un puntero a la primera celda de memoria asignada.

Se ha de ser cuidadoso en dos puntos: cuando los grupos hoja G_i del grupo G sean eliminados de la jerarquía de grupo actual, entonces todos los segmentos de memoria compartida de los G_i se unifican para formar el almacenamiento libre del grupo G. Esta decisión de diseño se ha hecho por razones de economía de espacio, sin embargo, esto implica que todos los datos asignados en uno de los *heaps compartidos* de los G_i son eliminados automáticamente (para ciertos algoritmos PRAM esto constituye un inconveniente pues la estructura dinámica de datos creada por los procesadores de los grupos hoja deberá ser copiada a un lugar seguro antes de que los grupos hoja sean eliminados de la jerarquía).

El segundo problema surge con los punteros a funciones. Primero nótese que en *fork95* a diferencia del lenguaje original, todos los valores devueltos por una función son privados. Entonces sh sync int (*f)(int) no denota un puntero a una función síncrona que devuelve un valor

compartido, sino un puntero compartido a una función síncrona. En el caso de un puntero compartido, todos los procesadores dentro del grupo hoja en cuestión ejecutarán la misma llamada y todo irá bien. Una llamada a una función a través de un puntero privado, sin embargo puede ser vista como una gran sentencia *switch*. Consecuentemente, se deben crear en el sistema grupos separados para cada función del tipo apropiado. Entonces el usuario tiene que, o bien determinar aquellos grupos que no son vacíos o bien debe proveer un espacio de almacenamiento separado para cada función apuntada. Como en general, la mayoría de las funciones no se invocan, la segunda alternativa puede provocar un tremendo gasto de memoria compartida y tamaño del programa. Esta es la razón por la que en la versión actual de *fork95*, las llamadas a función a través de punteros privados se ejecutan asíncronamente. Técnicamente esto se consigue mediante su inclusión implícita en una sentencia *farm* supuesto que no estén ya situadas dentro de una, o bien que se trate de una función calificada como *async*. De acuerdo con esto, los punteros privados deberían sólo apuntar a funciones calificadas como asíncronas.

3.2.3 Saltos controlados.

En *fork95* se distinguen cincos formas distintas de saltos:

1. **goto l** salta a la sentencia etiquetada con l.
2. **continue** salta al final del cuerpo del bucle incluido más profundamente.
3. **break** salta al final de la sentencia de bucle o switch que lo incluya más profundamente.
4. **return** salta al final del cuerpo de la función actual
5. **exit** salta al final del programa

Al menos en este contexto, se considerará a las sentencias *start* y *farm* como bucles especiales (llamados paralelos). Sobre la sentencia fork no se hace ninguna consideración especial. La razón de esta decisión fue el extender la analogía entre la sentencia *fork* y la sentencia *if* (cuando la

condición depende de variables privadas), puesto que ambas sentencias crean nuevos subgrupos y distribuyen procesadores. Por lo tanto, en *fork95* las dos sentencias tendrán un comportamiento similar con respecto al *break* y al *continue*.

En un ámbito paralelo, los saltos pueden ser incluso más inconvenientes que en el ambiente secuencial:

- Mediante un salto fuera de una sentencia *start* un procesador podría escapar de ser desactivado al final del cuerpo.
- Mediante un salto fuera de una sentencia *fork* o una sentencia condicional que dependa de datos privados un procesador podría salir del grupo al que pertenece. De acuerdo con esto, si se salta al cuerpo de otro *fork* este podría entrar a formar parte de otro grupo de manera irregular.

Está absolutamente sin clarificar cuál sería el significado de que un procesador atravesara la frontera de una sentencia *farm*.

El problema que surge en cualquiera de estas situaciones es que el número de procesadores dentro de un grupo y por ende el número de procesadores involucrados en, por ejemplo, operaciones de sincronización para ese grupo puede cambiar de manera impredecible.

En el estado actual de la definición del lenguaje y su implementación hay que tener especial cuidado y sólo utilizar los saltos *continue*, *break* y *return*. En estos casos, la dirección del salto es “desde dentro hacia fuera” hacia el final de alguna sentencia abarcadora. Especialmente, el número de grupos abarcadores que construyen sentencias (esto es, la longitud del camino en la jerarquía de grupo desde el grupo hoja actual al grupo hoja en el destino del salto) se puede determinar en tiempo de compilación. Por lo tanto, se pueden generar las actualizaciones correspondiente para volver a computar el número de procesadores remanentes de todos los grupos implicados. Nótese sin embargo, que se ha de añadir una sincronización extra en el código correspondiente a la traducción de bucles que contengan *continue* o *break*. En el caso de *break*, la sincronización ha de ser insertada

inmediatamente después del bucle y en el caso del *continue* inmediatamente después del cuerpo del bucle. Análogamente, la ejecución de funciones que contienen *return* debe ser finalizada mediante una sincronización.

3.3 Un Ejemplo usando la Técnica Divide y Vencerás.

Como ilustración de la potencia del concepto de grupo, se proporciona en el Código 12 la implementación en *fork95* de un algoritmo de ordenación [Bha]. Dicho algortimo se basa en la división implícita en grupos mediante el uso de la sentencia if. Este ejemplo se utilizará en el capítulo siguiente para comparar las características de FORK, *fork95* y ll.

```
1    #include <fork.h>
2    #include <assert.h>
3    #include <io.h>
4    #define Nmax
5    sync void quicksort( sh int *);
6    pr int x, pos = 0;
7    sh int a[];
8    main(){
9    start {      /* se inicializan todos los procesadores disponibles */
10     sh int numElem;
11     x = a[];
12     quicksort( &numElemt );
13     a[pos-1] = x;
14    /* start */
15   /* main */
16   sync void quicksort( sh int *numElement) {
17    sh int pivot, numLeft = 0, numRight = 0;
18    pr int left, right;
19    pivot = x;
20    left =  (x < pivot);
21    right = (x > pivot);
22    if (x != pivot)
23       if (left)  quicksort( &numLeft );
24       else     quicksort( &numRight );
25    if (x == pivot)  pos = numLeft + 1;
26    if (right) pos += 1 + numRight;
27    *numElement = numLeft + numRight + 1;
28   } /* quicksort */
```

Código 12. Un Algoritmo de Ordenación en fork95.

Capítulo III

El Lenguaje ll

1. El Sistema ll.

El sistema ll representa un esfuerzo encaminado a integrar las características deseables en un lenguaje de programación orientado al modelo PRAM, expuestas en la introducción del segundo capítulo. El lenguaje ll, es una extensión de un subconjunto de Pascal, que ofrece la posibilidad de expresar paralelismo controlando los puntos de sincronización y comunicación a través de un espacio de direcciones compartido y permitiendo implementar de forma sencilla y directa algoritmos paralelos recursivos. En el sistema ll, la red de interconexión no es visible directamente desde el lenguaje. No hay paso de mensajes explícito, en su lugar se admiten lecturas y escrituras de la memoria compartida. Además, ll dispone de mecanismos para relajar la sincronización que lo hacen ideal para la

descripción de algoritmos en los que el grano de la sincronización es grueso [Val90a].

Desde 1989, fecha en que se construyó el primer prototipo, ll se ha utilizado en la enseñanza de algoritmos paralelos en la asignatura de Ciencias de la Computación II en quinto curso de la licenciatura en Ciencias Matemáticas y en las asignaturas de Programación en Paralelo I y Programación en Paralelo II en quinto curso de la Ingeniería Superior de Informática y en cursos de Tercer Ciclo (doctorado) del Programa de Estadística, Investigación Operativa y Computación. Así mismo, ha sido utilizado por el Grupo de Computación en Paralelo de la misma Universidad para sus labores de investigación, como herramienta de diseño y verificación de algoritmos paralelos. Tenemos noticias de que el sistema ll ha sido utilizado en Polonia, en la Universidad de Varsovia, donde ha sido introducido por el grupo de trabajo del profesor Rytter [Gib88].

La primera versión del lenguaje y su compilador se terminaron a finales de 1991 constituyendo el núcleo de la memoria de licenciatura que precedió a este trabajo y que se leyó en 1992 [Leo92]. Una versión para redes de transputers fue el resultado de un segundo trabajo de licenciatura en 1993 [San93]. Haciendo uso de ll como herramienta de diseño de algoritmos PRAM, se han realizado cuatro memorias de diplomatura [Hid93, Mar93b, Nie93] y una memoria de licenciatura [Mar93a]. Así mismo, se han desarrollado trabajos sobre diferentes variaciones del lenguaje y su implementación en las memorias de diplomatura [Cha93, Her94].

El lenguaje ha ido evolucionando conforme se adquiría experiencia en los aspectos del diseño de algoritmos PRAM y en la implementación de lenguajes. Actualmente, existen tres implementaciones correspondientes a tres versiones distintas del lenguaje que naturalmente se corresponden a diferentes grados de madurez. La versión más antigua data de 1989-1990 y se ejecuta sobre un simulador que corre sobre máquinas IBM-PC compatibles (aunque su transporte a cualquier otra plataforma es trivial). Esta versión dispone de paralelismo anidado, virtualización de

procesadores, posibilidad de mezclar paralelismo y recursividad y la instrucción de llamada paralela a subprogramas. En la segunda versión, que data de 1993, aparecen como elementos más notables la diferenciación entre variables compartidas y privadas, así como el cualificador RELAX para desactivar la sincronización. Esta versión corre sobre un intérprete que se ejecuta sobre una red de transputers. Una tercera versión de ll (1995), la última de las que han sido implementadas, produce una mezcla de código occam y ensamblador para transputers. De este modo se mejora el rendimiento del sistema. Aún cuando las versiones no son totalmente compatibles, la transformación de un programa de una versión a otra es inmediata. En este capítulo se describe la experiencia acumulada durante estos siete años de programación e investigación.

2. Creando un Ejecutable.

En la versión 1.0 del compilador ll se genera código para una máquina PRAM orientada a pila (Figura 1). Este código desciende directamente del P-código utilizado en [Han85], ampliado con nuevos operadores que se corresponden de manera directa con las nuevas construcciones de alto nivel. En la primera versión del sistema ll se considera al modelo PRAM una máquina SIMD. Este punto de vista, fue evolucionando a una SPMD en las versiones 2.0 y 3.0 del sistema. Seguiremos aquí una descripción “histórica”, introduciendo el modelo en su concepción inicial para irlo ampliando conforme se presentan las construcciones del lenguaje.

Con relación a la máquina PRAM de la Figura 1, nótese que el *procesador Control* dispone de un puntero de instrucción (IP) que apunta en cada instante a la instrucción que se ha de ejecutar. Se dispone de un registro de control (C) que indica qué procesadores están activos y cuáles permanecen ociosos en el ciclo de instrucción en curso. Un puntero de pila (SP) que apunta a la memoria de trabajo del control. Por último, un puntero

de base (BP) que se utiliza para hacer referencias en el registro de activación en curso.

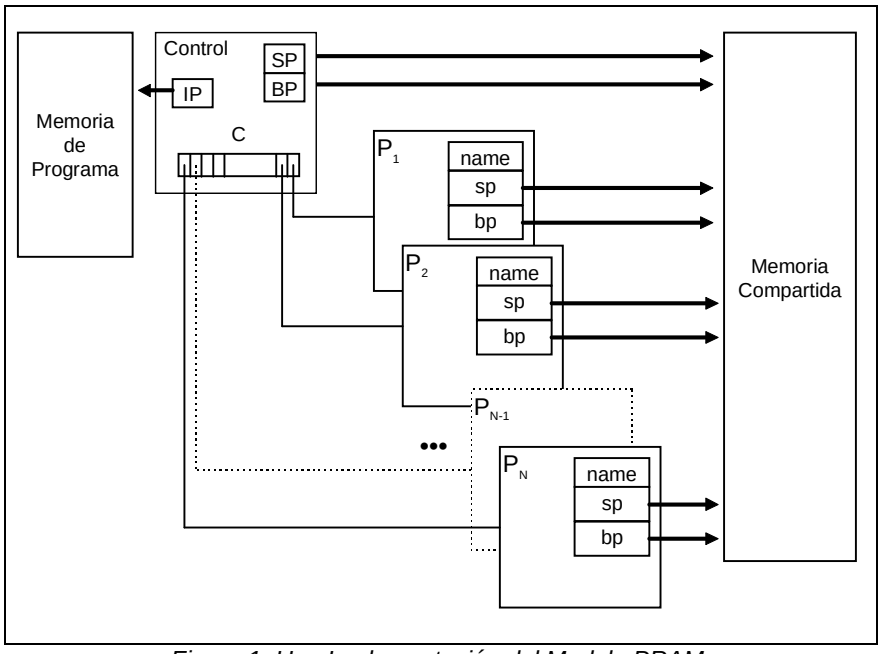


Figura 1. Una Implementación del Modelo PRAM.

Cada uno de los *procesadores elementales* dispone de un puntero de pila (sp), un puntero de base (bp) y un registro que guarda el nombre lógico del procesador elemental (name). Sin embargo, en esta versión, los procesadores elementales, no disponen de contadores de programa.

La memoria se representa mediante un array de valores enteros. A los elementos del array los denominaremos *palabras* y a sus índices *direcciones*. Se distinguen dos tipos de memoria: *memoria de programa* y *memoria de datos*. La memoria de programa contiene las instrucciones que ejecutan los diferentes procesadores, se accede a ella mediante el registro IP del procesador Control. La memoria de datos de cada procesador elemental y del procesador control se gestiona como una pila utilizando los registros sp y SP respectivamente.

En cada ciclo de reloj, el procesador Control, carga la instrucción de la memoria de programa apuntada por el registro contador de programa

(IP). Si la instrucción es de transferencia de control la ejecuta sin que intervengan los procesadores elementales. Si se trata de otro tipo de instrucción, se emite en paralelo a cada uno de los procesadores activos, los cuales síncronamente ejecutan la instrucción. A continuación el procesador Control incrementa el contador de programa y el ciclo se repite.

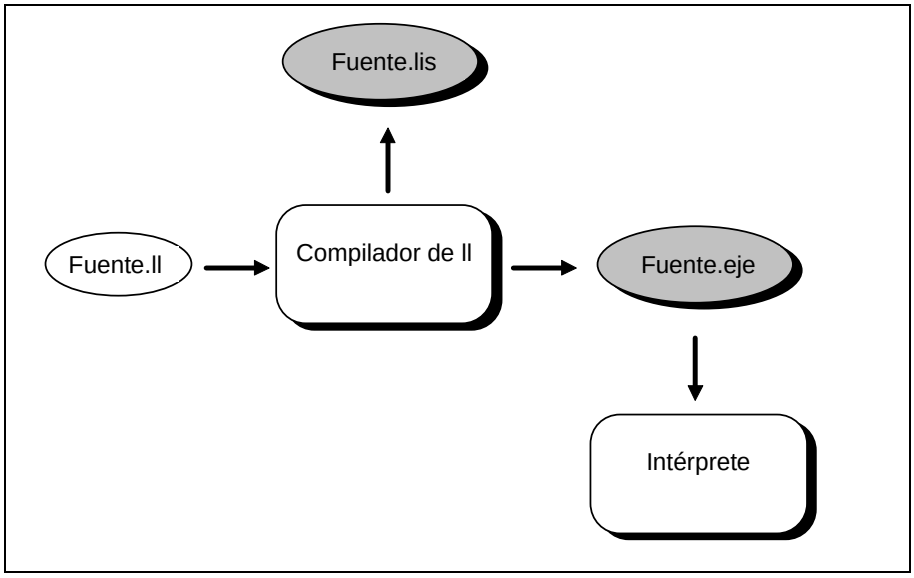


Figura 2. Proceso de Compilación para las Versiones 1.0 y 2.0 .

El sistema ll en su primera versión [Leo93] cuenta con un intérprete de la máquina descrita para IBM-PC compatibles y en la segunda con un intérprete sobre una red de transputers. En ambas versiones el compilador genera dos ficheros, uno con extensión ".eje" que le sirve como entrada al intérprete, y otro con extensión ".lis" que contiene el código fuente numerado junto con el código ensamblador correspondiente a las sentencias del programa fuente. Este último fichero se puede utilizar como referencia para depurar programas (Figura 2).

La última versión (3.0) del compilador ll genera dos ficheros occam [Inm84]: SEQINS.INC y PARINS.INC que contienen la traducción de las partes secuenciales y paralelas respectivamente, del código fuente. Estos ficheros son la parte variable del conjunto de ficheros occam que constituyen esta

versión del sistema ll. Usando el Occam Toolset D7205 [Inm91], el conjunto completo de ficheros occam se compila para producir el código final para la red de transputers. Para obtener más flexibilidad, un programa adaptador permite al programador modificar algunas características del entorno tales como la topología de la red de transputers, la jerarquía de procesadores, la cantidad de memoria compartida, el número de procesadores virtuales necesarios, etc. En consecuencia, el sistema ll consta de un esqueleto escrito en occam donde se incluyen los ficheros SEQINS.INC y PARINS.INC producidos por el compilador (Figura 3). La sincronía inicial del lenguaje se debilitó en las versiones 2.0 y 3.0 para permitir una ejecución más eficiente en máquinas MIMD. Ahora el compilador tiene la tarea adicional de controlar la sincronización con la ayuda del hardware y/o de librerías de rutinas para tiempo de ejecución.

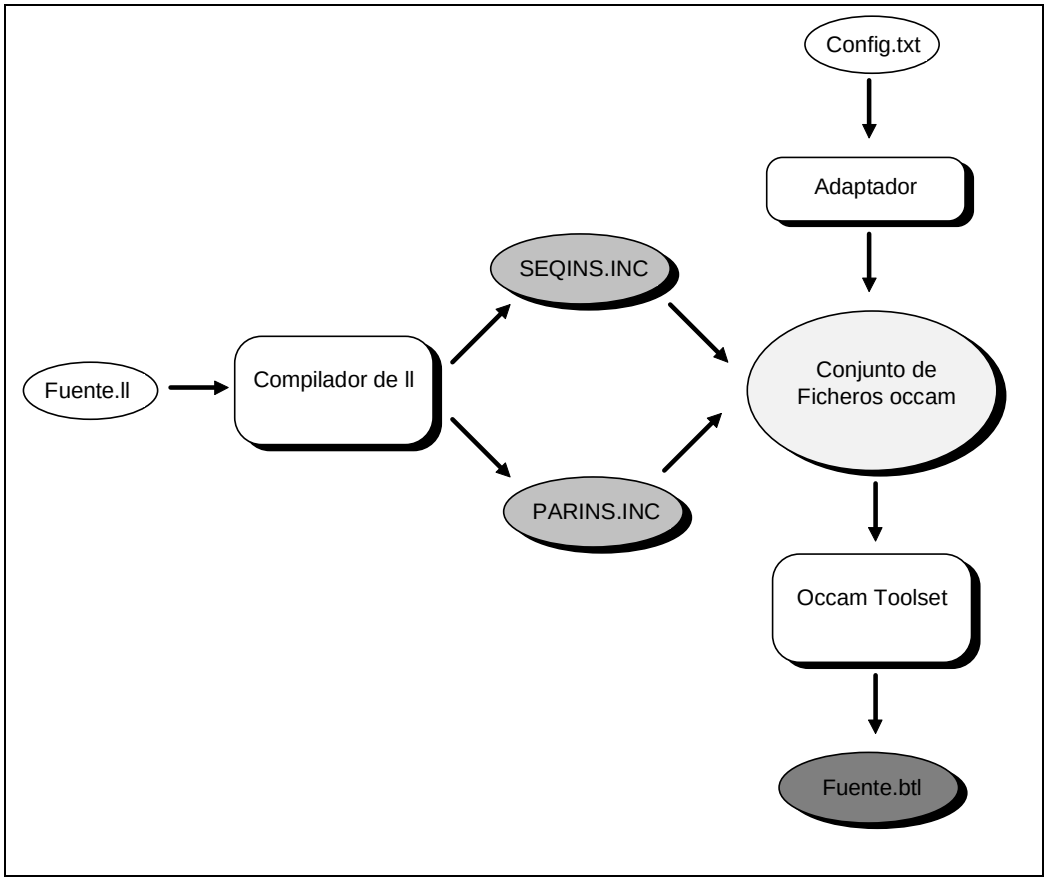


Figura 3. Creación de un Ejecutable.

El P-código extendido generado por las primeras versiones del compilador se utiliza como lenguaje intermedio para la última implementación. Para describir la traducción de las sentencias del lenguaje para la red de transputers que constituye la máquina objeto del compilador, se usarán definiciones dirigidas por la sintaxis que hacen uso de tales operaciones intermedias. En la siguiente sección se describe el comportamiento de las operaciones más importantes.

3. El Lenguaje Intermedio.

En este apartado se describe el lenguaje intermedio que se utiliza en la traducción del lenguaje ll. La notación intermedia en sus orígenes estaba orientada a expresiones, permitiendo una pareja de operandos por operador. Para ensanchar el rango de utilidad, se amplía el conjunto de operaciones permitidas y el número de operandos asociados a cada operador puede ser variable. En [Leo92] se encuentra una explicación detallada de la primera ampliación realizada (páginas 8-24). A continuación se describen de forma abreviada las operaciones del lenguaje intermedio final. Una explicación pormenorizada de las mismas se encuentra en [San93] (páginas 173-221).

Las instrucciones del lenguaje intermedio tienen la forma general:

$$OP \quad Arg_1 \quad Arg_2 \quad ... \quad Arg_n$$

El operador n-ario anterior tiene su primer argumento situado n-1 posiciones por debajo del tope de la pila y su argumento n-ésimo en la cima de la pila. La evaluación aplica el operador (OP) a los *n* valores que se encuentran en la cima de la pila, extrae los operandos e inserta el resultado en la pila.

Las instrucciones de la lista de operadores (OP) que aparecen en la Tabla I no tienen operandos y tienen la interpretación estándar de las operaciones aritméticas, lógicas y de desplazamiento de bits.

ADD	DIVIDE	MINUS	MAX	MIN
MODULO	MULTIPLY	SUBTRACT	AND	EQUAL
GREATER	LESS	NOT	NOTEQUAL	NOTGREATER
NOTLESS	OR	XOR	SHL	SHR

Tabla I. Instrucciones para Expresiones.

Las instrucciones para los operadores que aparecen en la Tabla II son las más comunes en la manipulación de la memoria.

VARIABLE level displacement	Coloca la dirección de una variable en el tope de la pila del procesador	Ejecución Paralela y secuencial
VALUE length	Retira la dirección que hay en el tope de la pila y deja en el tope el valor almacenado en ella	Ejecución Paralela y secuencial
ASSING length	Se sacan de la pila la dirección de la variable y el valor que se desea asignar, y se almacena en la dirección el valor correspondiente	Ejecución Paralela y secuencial
CONSTANT value	Coloca la constante value en la cima de la pila del procesador	Ejecución Paralela y secuencial

Tabla II. Instrucciones de Manipulación de Memoria.

Las instrucciones de salto (Tabla III) permiten la modificación del flujo de control del programa a diferentes puntos del mismo.

Las instrucciones de control del paralelismo (Tabla IV) controlan el número de procesadores elementales activos en cada momento de la ejecución.

En los apartados que siguen se presenta en detalle el sistema ll. Se describen las sentencias del lenguaje así como la semántica y la traducción que se emite en la versión actual para un multicomputador. Al especificar la traducción a lenguaje intermedio de cada sentencia, se precisa el significado de las instrucciones introducidas y catalogadas en esta sección.

DO displacement	Bifurca si hay un cero en el top de la pila del procesador control	Ejecución secuencial
GOTO displacement	Bifurca incondicioanlmente a la dirección IP + displacement	Ejecución secuencial
PROCCALL level displacement	Reserva espacio para el registro de activación. Almacena el enlace estático, el dinámico, la dirección de retorno y salta al comienzo el procedimiento	Ejecución Paralela y secuencial
PROCEDURE var.length control.stack elem.stack displacement	Reserva espacio en el registro de activación para las variables locales y coloca el IP en la primera instrucción del subprograma.	Ejecución Paralela y secuencial
ENDPROC param.length	Saca de la pila en tiempo de ejecución el registro de activación del subprograma que finaliza y coloca en IP la dirección de retorno.	Ejecución Paralela y secuencial
GOIFZERO displacement	Bifurca a IP + displacement si no es cero el registro de control (C)	Ejecución Paralela
GOIFNOZERO displacement	Bifurca a IP + displacement si no cero el registro de control (C)	Ejecución Paralela

Tabla III. Instrucciones de Bifurcación.

PATTERN displacement	Establece un patrón de procesadores activos. Si el rango a activar es negativo, se salta a IP+ displacement	Ejecución secuencial
NESTEDPATTERN displacement	Establece un patrón de procesadores activos. Si el rango a activar es negativo, se salta a IP+ displacement	Ejecución Paralela
OLDPATTERN	Recupera el antiguo patrón de procesadores activos.	Ejecución Paralela y secuencial
PUSHPATTERN	Inserta el patrón actual de procesadores activos de la pila del procesador de control	Ejecución Paralela
POPPATTERN	Recupera el antiguo patrón de procesadores activos de la pila del procesador de control	Ejecución Paralela
COPYTOPS	Se coloca en el correspondiente bit del registro de control un cero o un uno dependiendo de los valores que se encuentren en los tops de las pilas de los procesadores elementales activos.	Ejecución Paralela
COMPLEMENT	Complementa el registro de control	Ejecución Paralela

Tabla IV. Instrucciones de Control de Paralelismo.

4. Generalidades acerca de la Implementación para una Red de Transputers.

Las versiones 2.0 y 3.0 del sistema ll proyectan el modelo PRAM sobre una red de transputers. La topología elegida para conectar los procesadores, se especifica en el fichero de configuración (*config.txt* en la Figura 3). Los procesadores son distribuidos según una jerarquía o árbol. El grafo de conexión de la red no tiene porqué adaptarse a la jerarquía determinada por el programador. Sin embargo, para la aproximación que se presenta en este trabajo, se fija el grafo de conexiones de los transputers a un árbol ternario (Figura 4). El grado del árbol viene impuesto por el número de enlaces (links) disponibles en un transputer. La topología de conexión elegida se adapta adecuadamente a la forma en la que los diferentes procesadores se activan y desactivan al ejecutarse las sentencias paralelas, lo que intuitivamente lleva a pensar en una ejecución más eficiente que con cualquier otra topología. Cada transputer se encarga de simular el comportamiento de un cierto número de procesadores elementales.

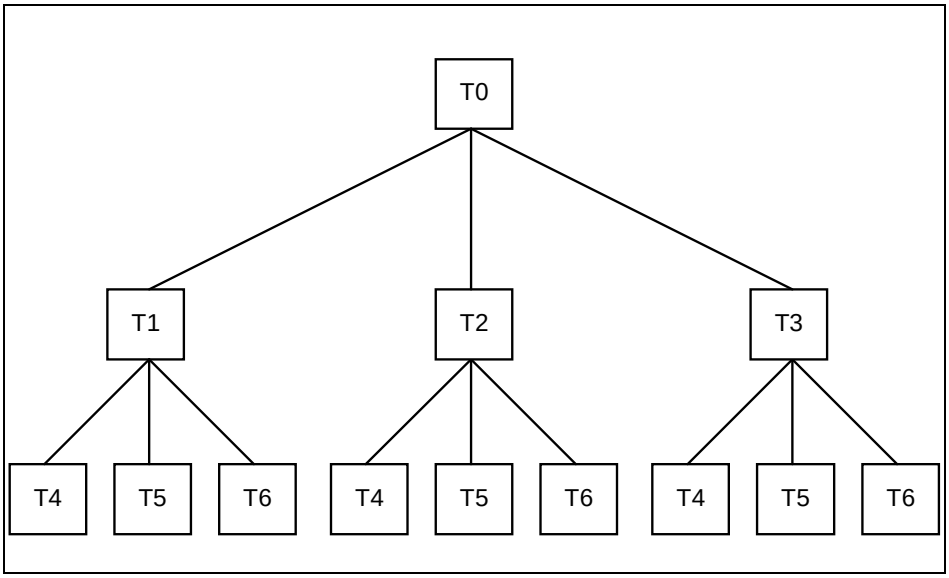


Figura 4. Topología de Árbol para la Red de Transputers.

La implementación de la PRAM sobre la red configurada según un árbol, constituye, en estas versiones, el *esqueleto* en el que se incluyen los ficheros occam generados por el compilador. Dicho esqueleto soporta rutinas que implementan en occam las acciones que tienen que llevar a cabo cada una de las instrucciones del lenguaje intermedio descrito superficialmente en la sección anterior.

Distinguiremos entre procesadores lógicos o virtuales (procesos) y procesadores físicos o reales (hardware). Un procesador físico tiene la capacidad de simular un cierto número de procesadores PRAM lógicos. El número de procesadores virtuales activos a lo largo de una computación en el sistema ll, varía entre uno (caso secuencial) y el número máximo de procesadores virtuales que pueda soportar la arquitectura, que es, en general, mayor que el número de procesadores físicos.

El proceso de virtualización de procesadores es esencial para obtener portabilidad y permitir una programación estructurada. Sin virtualización es complicado implementar algoritmos que requieran un número elevado de procesadores. Uno de los inconvenientes de fork95 es la ausencia de virtualización: Se produce un error en tiempo de ejecución si una sentencia de activación de procesadores requiere un número mayor que el número de procesadores disponibles. En general, la situación ideal es que la virtualización de procesadores sea soportada por el hardware en lugar de dejarla a cargo del software.

Tanto los procesadores virtuales como los transputers tienen tres estados posibles: *Activos*, *Inactivos* y en *Espera* de recibir trabajo.

El estado de un transputer se define en función del estado de los procesadores elementales que virtualiza: estará inactivo cuando todos los procesadores que virtualiza lo están. Se encontrará en estado de actividad cuando alguno de los procesadores que virtualiza está activo. El estado de espera de recibir trabajo por parte de un transputer se da cuando también están en ese estado todos sus procesadores virtuales.

Un procesador elemental está en espera de recibir trabajo cuando el Control de la máquina no ha decidido que intervenga en la ejecución de instrucciones. Es posible, que un procesador que en un instante dado esté en espera de recibir trabajo ya haya intervenido previamente ejecutando cierta parte del código del programa, acabada la ejecución de la cual pasó al estado de espera en el que se encuentra. Los procesadores elementales estarán activos en el momento en que se encuentren ejecutando las instrucciones del programa. Su estado de inactividad se produce cuando de forma momentánea se inhiben de la ejecución de ciertas partes del código.

En el caso de los transputers es necesario hablar de un cuarto estado, el estado de *sincronización*, que tiene lugar cuando procesadores que el transputer en cuestión virtualiza no están ejecutando instrucciones, sino que son los procesadores virtuales situados en el transputer descendiente del considerado los que están trabajando.

Al comienzo de la ejecución de cualquier programa, el transputer raíz difunde una copia del programa a través de toda la red de transputers, y acomete la ejecución del mismo. Si el código es estrictamente secuencial, éste será ejecutado enteramente en el transputer raíz por el procesador control, mientras que si conlleva paralelismo, el transputer raíz activará el número de procesadores elementales necesarios; esta activación de procesadores lógicos puede implicar o no la activación de procesadores físicos: si el transputer en cuestión tiene un valor de umbral mayor que el número de procesadores lógicos necesarios, lo hace; si no, emite una señal de activación hacia sus transputers “hijos”. Este modo de operar se repite para todos los transputers de la red: si en un determinado transputer intermedio tiene lugar una demanda de procesadores lógicos, el nodo tiene la posibilidad de acometer la carga o bien traspasar carga a sus descendientes. Cuando el trabajo llega a un transputer “hoja”, éste solo puede optar o bien por acometer todo el trabajo o bien por generar una condición de error.

La estructura de los ficheros generados por el compilador del sistema ll, que se han de incluir en el sistema general de librerías que implementan la máquina descrita, es la que se muestra en la Figura 5.

El fichero secuencial estará constituido por una secuencia de código occam y llamadas a las rutinas que implementan las instrucciones que se han de ejecutar de forma secuencial.

El fichero paralelo es una tabla de decisión. Las etiquetas *code.number* permiten implementar modificaciones en el flujo de control paralelo. El conjunto de acciones (*actions*) representa, al igual que en el caso secuencial, el código occam y las llamadas a las rutinas del lenguaje intermedio que se han de ejecutar.

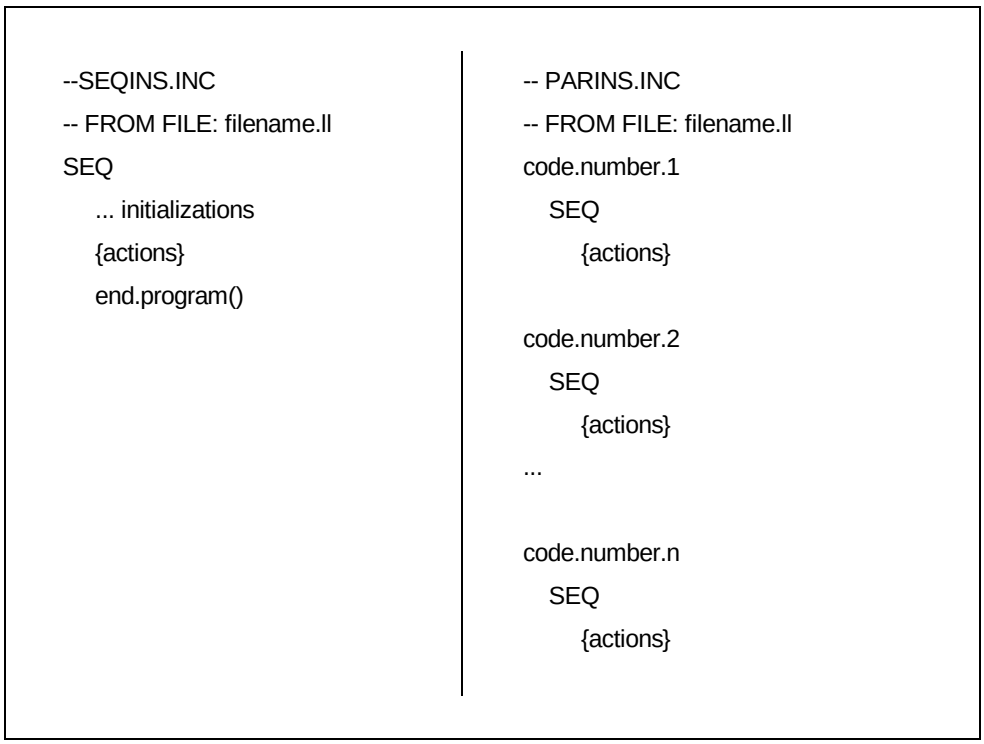


Figura 5. Esquema de los Ficheros SEQINS.INC y PARINS.INC.

5. El Lenguaje ll.

El lenguaje ll, es una extensión de un subconjunto de Pascal [Han85], que ofrece la posibilidad de expresar paralelismo. El incremento del número

de procesadores activos se produce como consecuencia de la ejecución de sentencias de activación de procesadores (*parallel*) mientras que el decremento se produce cuando finaliza la ejecución del cuerpo de estas sentencias. A continuación, se presentan cada una de las sentencias del lenguaje describiendo su sintaxis, semántica y proporcionando un esquema de traducción basado en el lenguaje intermedio.

5.1 Declaraciones.

En ll, existen dos tipos claramente diferenciados de almacenamiento de datos: el almacenamiento compartido y el almacenamiento privado. Los datos privados se almacenan en la memoria local del procesador al cual pertenecen. Los procesadores activos en el momento de la declaración de una variable crean múltiples instancias de la variable. Si se utiliza el cualificador **SHARED** en la declaración, la variable será compartida por todos los procesadores que sean activados dentro de su ámbito. Si no se utiliza ningún cualificador en la declaración se asume la declaración de una variable privada.

Si un conjunto de procesadores define una variable compartida *x* se crea una instancia de la variable *x* por cada procesador. Cada instancia puede ser accedida solamente por aquellos procesadores que sean activados por el procesador que creó dicha instancia. En contraposición, si un conjunto de procesadores define una variable privada *x* entonces cada procesador *p* del conjunto crea una instancia distinta de *x* la cual no es accesible a los descendientes de *p* en el árbol de activación.

En principio, los únicos tipos de datos simples que se admiten en la implementación actual del compilador, son el entero y el boolean. Así mismo, se pueden hacer declaraciones de los tipos estructurados arrays y registros. La implementación actual no soporta punteros, pero no existe ningún impedimento de diseño para que así no ocurra. En el apéndice A se puede encontrar una gramática completa de la versión 3.0 del lenguaje ll.

5.2 Activación de Procesadores: Sentencia Parallel.

La sentencia parallel en ll activa un conjunto de procesadores virtuales, todos ellos trabajando de forma concurrente. La sintaxis de la sentencia paralela es:

PARALLEL <Expresión1> **..** <Expresión2> **DO**
<Sentencia>

Cada procesador tiene asociado un identificador entero, ***NAME***, que permite identificarlo de forma unívoca.

Para definir la semántica de una sentencia *parallel*, consideremos el conjunto $R = \{a_0,\ldots,a_m\}$ de procesadores activos en el momento de su ejecución. Se designa por *l(s)* y *k(s)* ($s \in \{0,\ldots,m\}$) el resultado que obtiene a_s al evaluar <Expresión1> y <Expresión2> respectivamente. Entonces, cuando cada procesador a_s ejecuta una sentencia paralela, tienen lugar los siguientes pasos:

1. Para cada *s*, se calcula el número de procesadores elementales que deben ser activados por el procesador a_s. Sea $n(s) = k(s) - l(s) + 1$ este número. Entonces este procesador activa un conjunto $V^s = \{v_0^s, v_1^s, \ldots, v_{n(s)-1}^s\}$, con *n(s)* procesadores virtuales. Cada procesador v_t^s en V^s tiene su identificador **NAME** inicializado a *l(s)* + *t*.
2. Después de que todos los procesadores en R hayan completado el primer paso, el conjunto $\bigcup_{s=0}^{m} V^s$ es el nuevo grupo de procesadores que realizan síncronamente las instrucciones en <Sentencia>. El número total de procesadores virtuales activos es: $\sum_{s=0}^{m} (k(s) - l(s) + 1)$.

3. Al final de la ejecución de <Sentencia> se restaura el antiguo conjunto de procesadores activos.

Los procesadores virtuales V^S pueden acceder al espacio de datos compartido asociado al procesador virtual a_S .

En las versiones 2.0 y 3.0 la sentencia paralela también se puede utilizar con el cualificador **THRESHOLD**. Su función es informar del número de procesadores virtuales que es recomendable utilizar por procesador físico. Este cualificador no tiene efecto sobre la semántica del programa, pero sí sobre su eficiencia. La sintaxis completa de una sentencia paralela es:

PARALLEL <Expresión1>**..**<Expresión2>
[**THRESHOLD** <Constante>] **DO** <Sentencia>

Consideremos una vez más, el conjunto de procesadores virtuales activos $R = \{a_0,...,a_m\}$. Sea $H = \{p_0,...,p_n\}$ (H por Hardware) el conjunto de procesadores físicos de la máquina objeto. Sea $p \in H$ un procesador físico. Fijemos un instante en la computación y denotemos por R_p el subconjunto de procesadores virtuales de R cuya simulación es asignada a p (puede ocurrir que para algún p el conjunto R_p sea vacío). Evidentemente, R es la unión de R_p con $p \in H$. El número de procesadores en R_p se puede modificar si se requieren más procesadores virtuales. Si esto ocurre, el conjunto de acciones que tiene lugar es el siguiente:

1. El procesador físico p calcula, como se ha indicado antes, el conjunto R'_p de procesadores requeridos por los procesadores $a_i \in R_p$ ($R'_p = \bigcup_{a_i \in R_p} V^i$).

2. Si el valor $|R'_p|$ es mayor o igual que el valor especificado por el umbral <Constante>, entonces p comparte la simulación de los R'_p

procesadores entre los procesadores ociosos de *H*. En otro caso, el procesador *p* asume la simulación de R'_p .

La implementación actual del sistema ll inicializa el valor del umbral a uno si no se especifica ningún valor.

La Figura 6 ilustra las acciones descritas en los párrafos anteriores para una sentencia *parallel* con umbral 5, cuando es ejecutada sobre una máquina con tres procesadores físicos $H = \{p_0, p_1, p_2\}$. Antes de la activación, están activos cuatro procesadores virtuales. El procesador p_0 esta ocioso (R_{p_0} es vacío), p_1 esta encargado de dos procesadores lógicos $R_{p_1} = \{a_0, a_1\}$, y p_2 está simulando otros dos procesadores lógicos $R_{p_2} = \{a_2, a_3\}$. Después de la activación, el procesador p_2 simula $n(2) + n(3) = 4$, es decir, el número total de procesadores virtuales R'_{p_2} demandados por p_2 puesto que la cantidad requerida es menor que el umbral. Al contrario, el procesador p_1 distribuye la simulación de R'_{p_1} de los 5 procesadores lógicos requeridos por R_{p_1}.

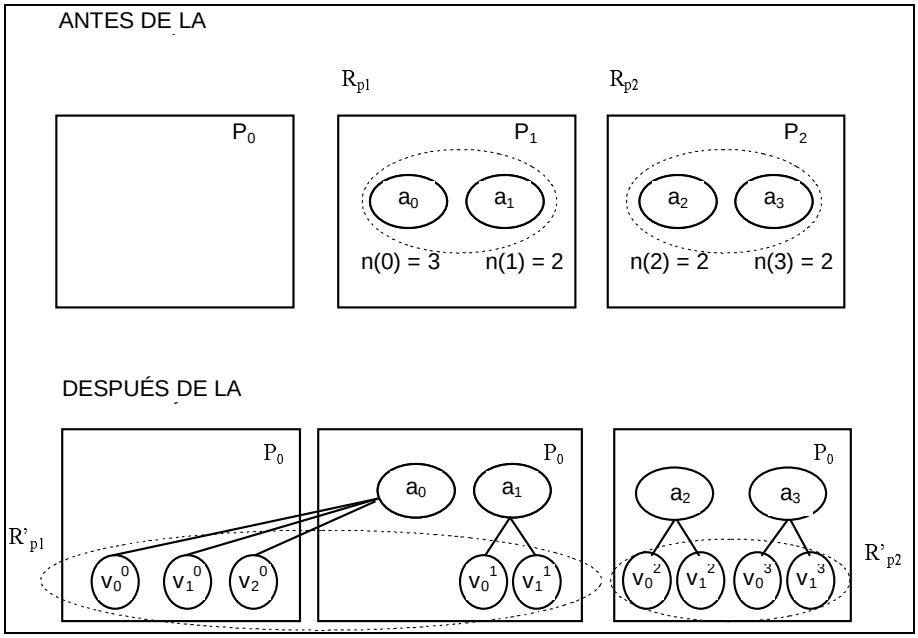


Figura 6. Activación de Procesadores Virtuales.

Ejemplo: Producto de Matrices.

Para clarificar la semántica de esta sentencia, consideremos el para el cálculo del producto de matrices de dimensión *NxN*. El ejemplo muestra el funcionamiento del anidamiento de sentencias de activación de procesadores.

La primera sentencia parallel (línea 5) activa *N* procesadores virtuales $V = \{v_1,...,v_N\}$. Cada uno de ellos ha de ejecutar la segunda sentencia parallel (línea 9), pero antes de ello almacena en la variable *row* el valor del identificador asociado al procesador (**NAME**). Nótese que la variable *row* ha sido declarada **SHARED** para que los procesadores que se activen posteriormente puedan tener acceso a ella.

El primer paso a ejecutar por la sentencia parallel de la línea 9 es determinar, para cada v_i , el número de procesadores virtuales que ha de activar a su vez cada procesador activo, $N_i = N$. Una vez que todos los procesadores activos llevan a cabo este cálculo, el nuevo conjunto de procesadores virtuales activos será $V' = \{v_1^1,...,v_N^1,v_1^2,...,v_N^2,...,v_1^N,...,v_N^N\}$ los cuales proceden a ejecutar la sentencia de asignación y el bucle de las líneas 12 y 13.

Cada procesador virtual de V' tiene definida una variable privada *sum* en la que se almacenarán los cálculos intermedios. Los procesadores virtuales de V' pueden acceder al espacio de datos compartido del procesador virtual que les activó. Al finalizar la segunda sentencia parallel se restablece el antiguo grupo de procesadores activos V .

□

```
 1  BEGIN
 2    readMatrix;
 3    setTime(t);
 4    RELAX
 5    PARALLEL 0..N-1 DO
 6    VAR row : SHARED INTEGER;
 7      BEGIN
 8        row := NAME;
 9        PARALLEL 0..N-1 DO
10         VAR sum : INTEGER;
11           BEGIN
12            sum := 0;
13             FOR 0 TO N-1 DO
14               sum := sum + A[row][COUNTER]*B[COUNTER][NAME];
15               C[row][NAME] := sum;
16           END;
17       END;
18     getTime(timer, t);
19     writeTime(timer);
20   END.
```

Código 1. Producto de Matrices en ll.

De la discusión anterior se desprende que al ejecutar una sentencia *parallel* se realiza una sincronización después de evaluar <Expresión2>.

El Esquema 1 muestra la traducción completa de una sentencia *parallel* a código intermedio. El indicativo **SYNC** en el esquema se utiliza para denotar la presencia de una sincronización en ese punto.

```
        Código <Expresión1>
        Código <Expresión2>
        SYNC                          -- Sincronización
        PATTERN CodeNumber Threshold endpat
                                      -- El procesador físico p almacena en su pila
                                      -- el conjunto de procesadores virtuales V
                                      -- El cual calcula el nuevo patrón
                                      -- de procesadores activos V',
                                      -- reequilibrando la carga de trabajo si
                                      -- es necesario.
                                      -- Si expresión2 <expresión1 se salta a endpat
        Código de <Sentencia>
endpat:                               -- Etiqueta
        ENDPATTERN                    -- Restablece el patrón de procesadores
                                      -- activos anterior, V.
        SYNC                          -- Sincronización
```

Esquema 1. Traducción de una Sentencia Parallel.

Ejemplo: Traducción de un ejemplo sencillo.

Para ilustrar el código final producido por el compilador para una sentencia de activación de procesadores, consideremos el ejemplo sencillo del Código 2. El código secuencial generado por el compilador se muestra en la Figura 7-(a) y el paralelo en la Figura 7-(b). Las sincronizaciones del esquema de traducción en este caso están implícitas en las llamadas a las rutinas *active.pattern* y *end.active.pattern*.

□

```
PROGRAM exampleOfparallel;
BEGIN
  PARALLEL 1..10 DO
  VAR a : integer;
    BEGIN
      a := 4;
    END;
END.
```

Código 2. Un Ejemplo Simple.

```
-- SEQINS.INC.
-- FROM FILE: parallel.ll
SEQ
  ... initializations
  --          CodeNumber, Threshold, exp1, exp2      --PARALLEL 1..10 DO
  active.pattern( 126,              1,      1,   10)
  end.program()
```

(a) Contenido del fichero SEQINS.INC

```
-- PARINS.INC.
-- FROM FILE: parallel.ll
126
  SEQ
    SEQ i = 0 FOR active.processors
      INT proc IS act[i]:
      INT bp IS b[proc]:
      SEQ
        par.newblock(1)            -- se reserva espacio para a
        mem[bp + (3)] := 4         -- a := 4
        par.endblock()             -- se libera el espacio de a
    end.active.pattern()
```

(b) Contenido del fichero PARINS.INC

Figura 7. Ficheros de Salida del Ejemplo Sencillo

Los procesadores de la red se estructuran según una jerarquía que se fija en tiempo de compilación. Cada procesador controla a un cierto subconjunto de procesadores descendientes. El pseudocódigo para la rutina de activación de procesadores aparece en la Figura 8. Después de computar cuántos procesadores virtuales son necesarios, se procede a comprobar si se supera la capacidad de simulación de la máquina paralela. En caso de que sea posible acometer el trabajo, se decide en términos del valor de **THRESHOLD** si la computación debe ser local o traspasada a otros procesadores. En el primer caso el procesador guarda la información de contexto sobre los procesadores virtuales previamente activos y acomete la ejecución. En el segundo caso, el procesador pasa a trabajar en tareas de sincronización. Los procesadores descendientes permanecen a la espera de una orden de activación procedente de su padre (Figura 9). Estos procesadores reciben el valor (*code.number*) en el que se indica el contador de programa en el que deben comenzar la ejecución, el número de procesadores solicitado y el registro de contexto. La información en el registro de contexto posibilita el acceso de los procesadores activados al espacio de datos compartido de sus ancestros en la jerarquía.

```
active.pattern(VAL INT code.number, threshold, BOOL active)
  SEQ
    ... Compute how many PRAM virtual processors are required
    IF
      total.procs.needed > max.processors.available
        ... error
      total.procs.needed > 0
        IF
          total.procs.needed <= threshold
            ... accomplish the work, active := TRUE
          TRUE
            SEQ
              ... activate other descendants starting with code.number
              state := synchronization.state
              synchronization()
      TRUE -- (total.procs.needed <= 0)
        ... jump to the end of parallel
:
```

Figura 8. Pseudocódigo de la Activación de Procesadores.

```
from.parent ? CASE
  activation; code.number; demanded; context.size::context
    tightly.synchronized := TRUE
  relaxed.activation; code.number; demanded; context.size::context
    tightly.synchronized := FALSE
  ... processor.activation
```

Figura 9. En Espera de Activación.

5.3 Expresiones y Sentencias de Asignación.

La sintaxis de una sentencia de asignación en ll es:

<Variable>**:=** <Expresión>

Para ejecutar una sentencia de asignación, el conjunto de procesadores virtuales activos lleva a cabo las siguientes acciones: se evalúa la expresión del lado derecho, se evalúa la dirección de la variable, se sincronizan los procesadores y se procede a almacenar en las direcciones correspondientes los valores calculados. Al final de la sentencia de asignación hay otro punto de sincronización. La traducción correspondiente sería la que se presenta en el Esquema 2.

```
Código de <Expresión>
Código para evaluar la dirección de <Variable>
SYNC
ASSING length
SYNC
```

Esquema 2. Traducción de una Sentencia de Asignación.

En general, el compilador intentará eliminar en la fase de optimización el mayor número de sincronizaciones mientras ello no afecte a la semántica del programa. El primer punto de sincronización se puede eliminar si se puede asegurar la independencia entre los lados derecho e izquierdo de la asignación. Por ejemplo, si se tiene una asignación como

A[NAME] := B[NAME] * C[NAME]

y es seguro que las variables *B* y *C* no coinciden con *A* (esto es, no son alias de *A*) se puede eliminar la primera sincronización. Sin embargo, en una sentencia como

A[NAME] :=A[2*NAME]

no se puede eliminar porque podría suceder; por ejemplo, que si el procesador número dos va más rápido que el uno, *A[1]* en lugar de contener *A[2]* termine conteniendo *A[4].*

Si se puede asegurar la independencia entre dos sentencias de asignación consecutivas, el compilador puede eliminar el segundo punto de sincronización. Consideremos por ejemplo el Código 3 , en la que el array *B* es privado: No son necesarias ninguna de las sincronizaciones al realizar la primera asignación y el compilador las eliminará.

```
VAR A: SHARED vector;
BEGIN
  PARALLEL 0..N-1 DO
  VAR B: vector;
  BEGIN
    B[NAME] := B[2 * NAME];
    A[NAME] := A[2 * NAME];
  END;
END
```

Código 3. Una Sentencia de Asignación en ll.

En la Figura 10 aparece el código generado por el compilador para el ejemplo del Código 3.

En la definición del lenguaje ll las operaciones de escritura concurrentes son resueltas de acuerdo con uno de los criterios especificados en el capítulo I mediante el uso de una directiva. Sin embargo, esta directiva no se ha implementado en ninguna de las versiones existentes. En la versión 1.0 (PCs) se utiliza el criterio *PRIORITY*, mientras en las versiones 2.0 y 3.0 se hace uso del criterio *COMMON*.

```
-- PARINS.INC.
-- FILE: test.ll
126
  SEQ
    SEQ i = 0 FOR active.processors
      INT proc IS act[i]:
      INT sp IS s[proc]:
      INT bp IS b[proc]:
      INT name IS name[proc]:
      SEQ
        par.newblock(10)                    -- se reserva espacio para B

        -- B[NAME] := B[2 * NAME];  una asignación PRIVADA (asíncrona)
        mem[((bp + (3)) + ((name - 1) * 1))] := mem[((bp + (3)) + (((2 * name) - 1) * 1))]

        -- A[NAME] := A[2 * NAME]; una asignación COMPARTIDA (síncrona)
        -- Valor de lado Derecho
        mem[sp + 1] := mem[((mem[bp] + (3)) + (((2 * name) - 1) * 1))]

   --ASSIGN
   assign.sync()                          -- SYNC
   SEQ i = 0 FOR active.processors
    ... Abreviations
    INT address :
    INT value :
    SEQ
      value := mem[sp + 1]
      address := ((mem[bp] + (3)) + ((name - 1) * 1))
      shared.store(address, value)
  assign.sync()                          -- SYNC

  SEQ i = 0 FOR active.processors
    par.endblock()
  end.active.pattern()                   -- final de la sentencia  parallel
```

Figura 10. Código Paralelo Generado para el Fuente del Código 3.

En las versiones 2.0 y 3.0 de ll las operaciones de escritura concurrentes del mismo valor en la misma dirección de memoria se consideran correctas. Cuando se asignan valores diferentes a la misma dirección de memoria, las diferentes replicaciones que constituyen dicha dirección (Figura 4 del capítulo I) contendrán alguno de esos valores diferentes. En este caso no se puede garantizar la coherencia de la dirección implicada.

```
VAR V : ARRAY[0..N-1] OF REAL;
VAR s: ARRAY [0..N-1] OF CARDINAL;
BEGIN
  FORALL i : [0..N-1] IN SYNC
    s[i] := 1;
    WHILE s[i] < N DO
      IF ((i MOD (s[i]*2))=0) AND ((i+s[i])<N) THEN
        V[i] :=  V[i] + V[i + s[i]]
      END (* IF *)
      s[i] := s[i] * 2
    END (* WHILE *);
  END (* sum in V[0] *)
END
```

Código 4. Suma de los Elementos de un Vector en Modula 2.*

La posibilidad de diferenciar de forma explícita entre variables privadas y compartidas permite una expresión clara de muchos algoritmos. Consideremos el ejemplo del Código 4 en el que aparece la implemetación en Modula 2* de la suma de los elementos de un vector, tomado de [Tic90] (página 12). La ausencia de variables privadas explíticas, fuerza al programador a declarar la variable *s* como un array, complicándole la tarea al compilador y al programador. Estos problemas se resuelven en el Código 5 escrito en ll para el mismo algoritmo, declarando la variable *s* como privada.

```
 VAR V : SHARED VECTOR;
BEGIN
  PARALLEL 0..N-1 DO
    VAR s : INTEGER;
    BEGIN
      s := 1;
      WHILE s < N DO
        BEGIN
          RELAX
            IF (NAME MOD s*2 = 0) AND (NAME+s < N) THEN
              V[NAME] :=  V[NAME] + V[NAME + s];
          s := s * 2
        END (* WHILE *);
    END (* PARALLEL *);
END
```

Código 5. Suma de los Elementos de un Vector en ll.

5.4 La Sentencia Compuesta.

Una sentencia compuesta en ll tiene la forma

BEGIN <Sentencia> {**;** <Sentencia>} **END**

La ejecución de cada <Sentencia> es síncrona, esto es, la <Sentencia>$_i$ debe ser terminada por todos los procesadores virtuales activos antes de que ninguno comience la ejecución la <Sentencia>$_{i+1}$.

A diferencia de Pascal, en ll es posible declarar variables locales a una sentencia compuesta, en tal caso, por cada procesador activo y por cada variable declarada se crea una réplica que es local a dicho procesador.

5.5 La Sentencia If.

Consideremos la forma general de una sentencia *if* :

IF <B> **THEN** <Sentencia1>
[**ELSE** <Sentencia2>]

Las acciones que tienen lugar cuando se ejecuta una sentencia condicional if-else, son las siguientes:

1. Consideremos el conjunto $R = \{a_0, a_1, \ldots, a_m\}$ de procesadores activos. Cada procesador de *R* evalúa las expresión <B>. Los valores de la condición dividen *R* en dos subconjuntos complementarios, $S = \{a_i \ / B_i = TRUE, i = 0, \ldots, m\}$ y $S' = \{a_i \ / B_i = FALSE, i = 0, \ldots, m\}$ donde B_i representa el resultado de la evaluación de <B> por el procesador *i*.

2. Después de que todos los procesadores en *R* han evaluado <B>, el conjunto de procesadores *S* para el que la evaluación de la condición ha resultado cierta, ejecutan <Sentencia1>. Cuando la ejecución de <Sentencia1> ha finalizado para todos los procesadores en *S*, los procesadores en *S'* ejecutan la <Sentencia2>.

3. Cuando todos los procesadores virtuales en *S'* finalizan la ejecución de <Sentencia2>, el conjunto de procesadores activos es el mismo que el del comienzo *R*.

Los pasos para ejecutar la sentencia if-then-else introducen dos puntos de sincronización, uno después de evaluar la condición <B> y otro antes de comenzar <Sentencia2>. Esto se aprecia con mayor claridad en el esquema de traducción utilizado por el compilador (Esquema 3). Recuérdese que V_p denota el conjunto de procesadores virtuales simulados por *p*.

```
        PUSHPATTERN                 -- Salvar el patrón de
                                    -- procesadores activos, Vp
        Código de <B>
        SYNC                        -- Sincronización
        COPYTOPS                    -- Decide el nuevo patrón de procesadores
                                    -- activos dependiendo de la evaluación de <B>
                                    -- S = Vp ∩ { vi / Bi = TRUE, i = 0,...,m }
        GOIFZERO else               -- Si S es vacío ...
        Código de <Sentencia1>
else:
        COMPLEMENT                  -- Complementa el patrón de procesadores activos
        SYNC                        -- Sincronización.
        GOIFZERO endif              -- Si S´ es vacío ...
        Código de <Sentencia2>
endif:
        POPPATTERN                  -- Se recupera el antiguo patrón, Vp.
```

Esquema 3. Traducción de una Sentencia If .

El número de procesadores activos se comporta como una pila: todos los programas comienzan y finalizan con un único procesador activo, y entre estos dos estados el número de procesadores activos varía de forma dinámica. Dado el comportamiento como una pila de la *activación de*

procesadores, se hace necesaria la presencia de instrucciones que posibiliten el almacenamiento y la recuperación del patrón de procesadores activos en cada instante. **PUSHPATTERN** y **POPPATTERN** son respectivamente estas instrucciones. En la Figura 11 se muestra el pseudocódigo correspondiente. El patrón de procesadores activo en cada instante se almacena en el registro de control, C, en el que cada bit indica el estado de un procesador virtual.

```
push.pattern()
   SEQ
      [mem FROM (sp + 1) FOR C.size] := C
      sp := sp + C.register.size
:

pop.pattern(INT active.processors)
   SEQ
      sp := sp - C.register.size
      C := [mem FROM (sp + 1) FOR C.size]
      active.counter(active.processors)
:
```

Figura 11. Almacenamiento y recuperación de procesadores activos.

La instrucción *push.pattern()* almacena el registro patrón de procesadores activos, C, en la memoria, mientras que *pop.pattern()* realiza la operación inversa, es decir, recupera en el registro C el patrón de procesadores almacenado en memoria. El procedimiento *active.counter()* actualiza la variable *num.active*, que almacena en cada instante el número de procesadores virtuales activos.

En la Figura 12 aparece el código correspondiente a las instrucciones **COPYTOPS** y **COMPLEMENT**. **COPYTOPS** decide cuáles de los procesadores activos ejecutarán <Sentencia1> dependiendo de los valores obtenidos de la evaluación de <B>. **COMPLEMENT** complementa el patrón de procesadores activos para que <Sentencia2> sea ejecutada por el grupo de procesadores que evalúan <B> como falsa.

```
copy.tops(INT active.processors)
  [MAX.C.REGISTER.SIZE] INT t:
  INT b:
  SEQ
    ... Initialization
    SEQ i = 0 FOR active.processors
      INT proc.name IS act [i]:
      SEQ
        b := mem[ s[proc.name]]
        ... set ith bit o t to b
        s[proc.name] := s[proc.name] - 1
    SEQ i = 0 FOR C.register.size
      C[i] := t[i]
    active.counter(active.processors)
:

complement(INT active.processors)
  SEQ
    SEQ i = 0 FOR C.register.size
      C[i] := C[i] >< mem[ sp - ((C.register.size - i) - 1)]
    active.counter(active.processors)
:
```

Figura 12. Instrucciones COPYTOPS y COMPLEMENT

5.6 Los Calificadores Relax y Calm.

El uso del calificador **RELAX** permite eliminar los puntos de sincronización dentro de una sentencia. Esto es, un procesador ejecutando código incluido en el ámbito de un cualificador **RELAX** no participará en ninguna sincronización y lo mismo es válido para todos los procesadores activados por él en el ámbito del RELAX. La sintaxis es:

RELAX <Sentencia>

El uso de **RELAX** permite mezclar partes de código síncronas y asíncronas. Esto constituye una ventaja cuando se le compara con otras soluciones, por ejemplo, en Modula 2* [Tic90] para conseguir el mismo efecto se tienen que desactivar los procesadores declarados para trabajar síncronamente (sync) y activarse para que trabajen asíncronamente (async), lo que supone un coste adicional de tiempo. Otra forma de solucionarlo en Modula 2* consistiría en declarar la lista de sentencias

totalmente asíncrona y utilizar semáforos para implementar las sincronizaciones, lo que supone un trabajo mucho más engorroso.

Muchas veces es necesario que el conjunto de procesadores que participan en una sentencia condicional :

IF B THEN S1 ELSE S2

para los que la evaluación de la condición ha resultado cierta, ejecuten síncronamente S1, el conjunto complementario ejecute síncronamente S2 y que los dos grupos de procesadores sean asíncronos entre sí. Esto puede conseguirse a través del uso del cualificador **CALM**, que elimina todos los puntos de sincronización excepto aquellos en el interior de sentencias de activación de procesadores que están dentro del ámbito del **CALM**. Esto es, un procesador ejecutando una sentencia en el ámbito de un **CALM** no participará en sincronizaciones, pero los procesadores activados por él o por sus descendientes en el árbol de activaciones sí lo hacen y constituyen un subgrupo síncrono. La sintaxis es:

CALM <Sentencia>

5.7 La Sentencias While y Repeat.

La sintaxis de la sentencia *while* es:

WHILE <B> **DO**
<Sentencia>

y es semánticamente equivalente a:

```
IF <B> THEN
  BEGIN
     <Sentencia>;
      WHILE <B> DO
          <Sentencia>
  END;
```

Por lo tanto, su semántica viene determinada por la de una sentencia condicional y la semántica de la recursividad. La traducción para una sentencia *while* aparece en el Esquema 4.

```
                PUSHPATTERN          -- Salvar el patrón de
                                     -- procesadores activos, Vp
beginwhile:
                Código de <B>
                COPYTOPS             -- Decide el nuevo patrón de procesadores
                                     -- activos dependiendo de la evaluación de <B>
                                     -- S = Vp ∩ { vi / Bi = TRUE, i = 0,...,m }
                SYNC                 -- Sincronización
                GOIFZERO endwhile    -- Si S es vacío ...
                Código de <Sentencia>
                SYNC                 -- Sincronización.
                GOTO beginwhile
endwhile:
                POPPATTERN           -- Se recupera el antiguo patrón, Vp.
```

Esquema 4. Traducción de una Sentencia While.

Análogamente, la sintaxis de la sentencia *repeat* es:

```
REPEAT
   <Sentencia>
UNTIL<B>
```

y es semánticamente equivalente a:

```
<Sentencia>;
WHILE <B> DO
   <Sentencia>
```

Por lo tanto, su semántica viene determinada por la semántica de una instrucción *while* y la semántica de la composición de sentencias. La traducción de esta sentencia aparece en el Esquema 5.

	PUSHPATTERN	-- Salvar el patrón de procesadores activos, V_p
beginrepeat:		-- Etiqueta de comienzo
	Código de <Sentencia>	
	SYNC	-- Sincronización
	Código de <B>	
	COPYNOTOPS	-- Decide el nuevo patrón de procesadores
		-- activos dependiendo de la evaluación de <B>
		-- $S = V_p \cap \{ v_i / B_i = FALSE, i = 0,...,m \}$
	SYNC	-- Sincronización.
	GOIFNOZERO beginrepeat	-- Si S es vacío ...
	POPPATTERN	-- Se recupera el antiguo patrón, V_p.

Esquema 5. Traducción de la Sentencia Repeat.

5.8 La Sentencia For.

La sentencia for tiene la sintaxis:

FOR <Expresión1> **TO** <Expresión2> **DO**
<Sentencia>

El contador viene dado por la variable predeclarada **COUNTER**. Se ha optado por esta opción heterodoxa para mantener la simetría con la sentencia *parallel*. La semántica de la instrucción for es equivalente a:

```
VAR
   COUNTER, final : INTEGER;
BEGIN
   COUNTER := <Expresión1>;
    final := <Expresión2> ;
    WHILE COUNTER <= final DO
       BEGIN
          <Sentencia>;
           COUNTER := COUNTER + 1;
       END;
END;
```

El Esquema 6 describe la traducción de esta sentencia.

```
        Código de COUNTER := <Expresión1>
        Código de final := <Expresión2>
        PUSHPATTERN                          -- Salvar el patrón de procesadores activos
beginfor:                                    -- Etiqueta de comienzo
        Código de  <COUNTER <= final >
        COPYTOPS                             -- Decide el nuevo patrón de procesadores
                                             -- activos
        SYNC                                 -- Sincronización
        GOIFZERO endfor                      -- Si el conjunto de proc. activos es vacío ...
        Código de  <Sentencia>
        Código de  <COUNTER := COUNTER + 1>
        SYNC                                 -- Sincronización.
        GOTO beginfor
endfor:
        POPPATTERN                           -- Se recupera el antiguo patrón.
```

Esquema 6. Traducción de una Sentencia For.

5.9 Los Subprogramas en II.

Hay dos partes esenciales en la traducción de subprogramas, el procesamiento de las declaraciones y el procesamiento de las llamadas. Primero el compilador encuentra las declaración de un procedimiento y dependiendo de si va a ser usado de forma paralela o no, lo traduce al código objeto adecuado. Porteriormente el compilador encontrará las llamadas, que se han de traducir como transferencias de control al código emitido durante la declaración.

5.9.1 Declaraciones de Subprogramas y de Tareas.

Las declaraciones de procedimientos son similares a las de Pascal. Su sintaxis es:

```
[SHARED | RELAXED] PROCEDURE <Identificador> [ (<Parámetros Formales>) ];
                        <Sentencia Compuesta>
```

Donde los parámetros formales, pueden ser definidos por referencia y por valor.

Si un procedimiento se va a llamar desde una sentencia paralela, esto es, va a ser ejecutado en paralelo, se ha de utilizar el calificador **SHARED** en su declaración. Es importante, tener esto en cuenta puesto que

la traducción de las sentencias no es la misma si han de ser ejecutadas por varios procesadores que si las tiene que ejecutar sólo uno. Distinguiremos por tanto entre "tareas" y "procedimientos". Esta distinción facilita la labor de generación de código por el compilador ll. El calificador **RELAXED** indica que además no son necesarias sincronizaciones dentro del procedimiento.

5.9.2 LLamadas a Procedimiento y LLamadas a Procedimiento Paralelas.

La sintaxis de una llamada a procedimiento es:

<Identificador> [**(** <Parámetros Actuales> **)**] **;**

Para definir el comportamiento de las operaciones que intervienen en la traducción de una llamada a procedimiento o tarea, se describe en primer lugar el entorno de ejecución. Un registro de activación (Figura 13) consta de los siguientes elementos:

- Un área de *parámetros* .
- Un área de *contexto* formada por:
  - El enlace estático.
  - El enlace dinámico.
  - La dirección de retorno.
- Un área de *variables locales.*
- Un área de *temporales* en la que se almacenan los operandos y resultados intermedios durante la ejecución de sentencias.

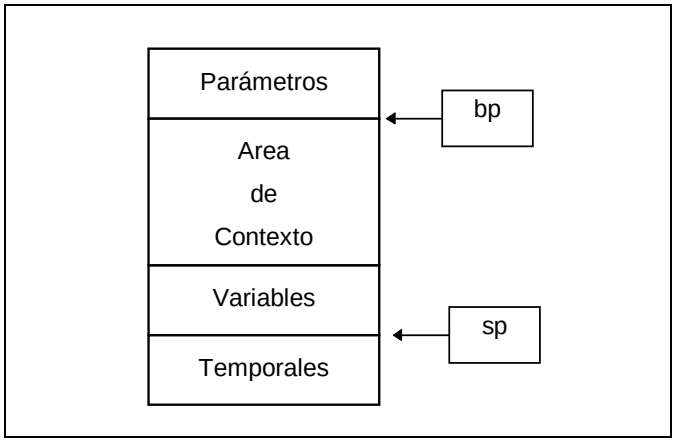


Figura 13. Organización de un registro de activación

Las acciones que componen una *secuencia de llamada* a procedimiento son las siguientes:

1. En primer lugar se reserva espacio en la pila para los parámetros actuales.
2. A continuación se almacenan los datos apropiados en el área de contexto. La instrucción de código intermedio encargada de realizar esta tarea es **PROCCALL**. El conjunto de acciones que tiene asociada esta instrucción son:
    - El enlace estático apuntará al registro de activación más reciente del bloque que contiene al bloque actual en el programa fuente.
    - El enlace dinámico contendrá el puntero de base del registro de activación que realiza la llamada.
    - En el campo reservado para la dirección de retorno, se almacena la dirección de la instrucción a la que se debe transferir el control de ejecución cuando finalice la ejecución del subprograma.
3. Después de crear la zona de contexto, la instrucción **PROCCALL** transfiere el control al código del procedimiento.
4. El procedimiento receptor de la llamada, se encarga de reservar espacio en la pila para los datos locales, y de bifurcar a la primera instrucción del

procedimiento. La instrucción del código intermedio **PROCEDURE** tiene como misión estas tareas.

La *secuencia de retorno* que ejecuta el procedimiento receptor de la llamada está implementada en la instrucción **ENDPROC**. Esta instrucción utiliza el enlace dinámico para obtener el registro de activación del procedimiento y bifurcar a la dirección de retorno.

Cuando un programa ejecuta una tarea en paralelo, su conjunto de sentencias tienen que ser ejecutadas en paralelo por los procesadores elementales que se encuentren activos. En este caso, cada uno de los procesadores activos ejecuta la instrucción **PROCCALL** y **PROCEDURE** que reservan el espacio adecuado en las pilas de los procesadores elementales y colocan los datos adecuados en el área de contexto.

Las acciones que tienen lugar cuando se llama a una tarea en modo secuencial son las siguientes:

- Se almacena el patrón secuencial en la pila: **PUSHPATTERN**
- Se activa un único procesador elemental: **SINGLECONTROL**
- Se emite la instrucción de llamada a subprograma **PROCCALL**
- Una vez finalizada la ejecución de la tarea por parte del procesador, se recupera el antiguo patrón secuencial: **OLDPATTERN**.

Existe en ll una ampliación del concepto de sentencia de llamada a procedimiento de Pascal que permite realizar múltiples llamadas a la misma subrutina en paralelo. La forma de la sentencia de llamada paralela en las versiones 2.0 y 3.0 es:

PARALLEL [**THRESHOLD** <Constante>] **DO**

$$t(x_1^0, x_2^0, \ldots, x_n^0) \| t(x_1^1, x_2^1, \ldots, x_n^1) \| \overset{k}{\ldots} \| t(x_1^n, x_2^n, \ldots, x_n^n)$$

---

donde t es un identificador de tarea (**SHARED PROCEDURE**) y los x_i^j denotan diferentes expresiones. La semántica de esta sentencia se explica a continuación:

Sea $A = \{a_0, a_1, \ldots, a_m\}$ el conjunto de procesadores virtuales activos que ejecutan la sentencia de llamada paralela. Para cada procesador virtual a_s tienen lugar las siguientes acciones:

1. Se activa un conjunto $V^s = \{v_0^s, v_1^s, \ldots, v_k^s\}$ de *k* procesadores. El número total de procesadores virtuales activos será por tanto de $k \times (m+1)$.
2. Cada uno de los procesadores recién activados v_i^s computa las expresiones $x_1^i, x_2^i, \ldots, x_n^i$, almacenando así los parámetros de la llamada en su pila.
3. Después de que todos los procesadores virtuales activos finalicen el paso 2, todos los procesadores activos ejecutan en paralelo la llamada a la tarea *t*.

5.9.3 Paso de Parámetros.

La sintaxis y la semántica del paso de parámetros en ll están basadas en las de Pascal. Las diferencias se establecen a partir del hecho de que los procesadores pueden acceder a dos clases diferentes de objetos: los compartidos y los privados. En la Tabla V se resumen las compatibilidades entre las declaraciones de los parámetros formales y los accesos con parámetros actuales, indicando que combinaciones están permitidas.

El uso como parámetros actuales de valores constantes, cuando los parámetros formales han sido declarados por referencia, produce un error al igual que en Pascal. Además se ha de tener en cuenta, que si un parámetro formal por referencia es compartido, el parámetro actual correspondiente no puede ser privado, pues su dirección de memoria se refiere a la zona de

datos privados y será interpretada como una dirección de la zona de datos compartida. De la misma forma, si un parámetro formal se declara por referencia privado y el parámetro actual correspondiente tiene tipo de almacenamiento compartido, se produce un error.

		Parámetros Actuales			
		POR REFERENCIA COMPARTIDO	POR REFERENCIA PRIVADO	POR VALOR COMPARTIDO	POR VALOR PRIVADO
Parámetros Formales	POR REFERENCIA COMPARTIDO	✓	✗	✗	✗
	POR REFERENCIA PRIVADO	✗	✓	✗	✗
	POR VALOR COMPARTIDO	✓	✓	✓	✓
	POR VALOR PRIVADO	✓	✓	✓	✓

Tabla V . Paso de Parámetros.

Consideremos las declaraciones ll del Código 6. La llamada al procedimiento compartido "*p*" es correcta pues los parámetros actuales "*a*" e "*i* " son compatibles con las declaraciones de los parámetros formales. Si se ulitiza como parámetro actual la variable "*b*" se produce un error puesto que la tarea "*p*" espera la dirección de una variable compartida como primer parámetro. El utilizar la variable "*j* " como segundo parámetro no tiene inconvenientes pues cada procesador reserva espacio en su zona de datos privada para el parámetro por valor "*y*" copiando en dicho espacio el valor correspondiente de "*j* ".

```
TYPE vector = ARRAY [1..N ] OF INTEGER;

VAR
    a : SHARED vector;
    b : vector;
    i  : INTEGER;
    j  : SHARED INTEGER;

 SHARED PROCEDURE p( VAR x : SHARED vector; y : INTEGER);
 BEGIN ...  END; (* p *)

  ...

 p(a, i);
  ...
```

Código 6. Ejemplo de Paso de Parámetros en ll.

6. Un Ejemplo de Programa ll.

Para explicar mejor la semántica de ll hemos seleccionado el problema del cálculo de las Sumas de Prefijos: sean *n* números almacenados en las componentes *A[0], A[1],..., A[n-1]* de un array *A*. El cálculo de la suma de prefijos consiste en hallar:

A[0], A[0] + A[1], A[0] + A[1]+ A[2],...

Para resolver este problema, utilizaremos el procedimiento recursivo del Código 7. El procedimiento *prefixSum* se declara usando el calificador **SHARED** para indicar que el procedimiento puede ser llamado simultáneamente por varios procesadores. En ll, se asume por defecto que las variables son privadas, por lo tanto es necesario declarar como compartidas (**SHARED**) las variables *left*, *right* y *mid* de las líneas 1 y 2, puesto que dichas variable son utilizadas dentro de una sentencia de activación de procesadores.

```
 1:  SHARED PROCEDURE prefixSum( left, right: SHARED INTEGER);
 2:  VAR mid: SHARED INTEGER;
 3:  BEGIN
 4:   IF left < right THEN
 5:     BEGIN
 6:       mid := (left + right) DIV 2;
 7:         PARALLEL DO
 8:           PrefixSum(left, mid) || PrefixSum(mid + 1, right);
 9:       PARALLEL (mid + 1)..right  DO
10:          A[NAME] := A[NAME] + A[mid];
11:     END
12:  END; (* PrefixSum *)
```

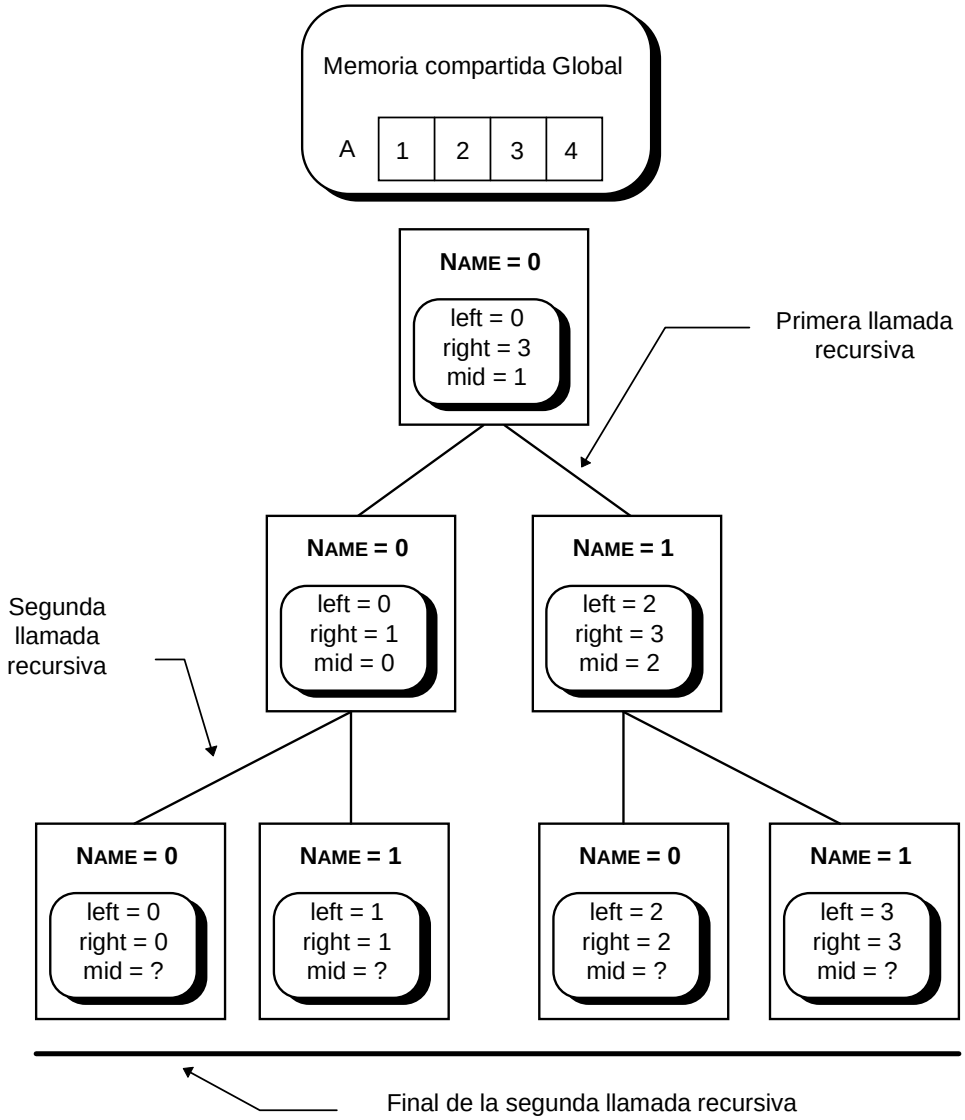
Código 7. Cálculo de la Suma de Prefijos en ll.

Las líneas 7 y 8 ilustran la posibilidad de mezclar recursividad y paralelismo en ll: se activan dos instancias del procedimiento *prefixSum* en paralelo con parámetros diferentes.

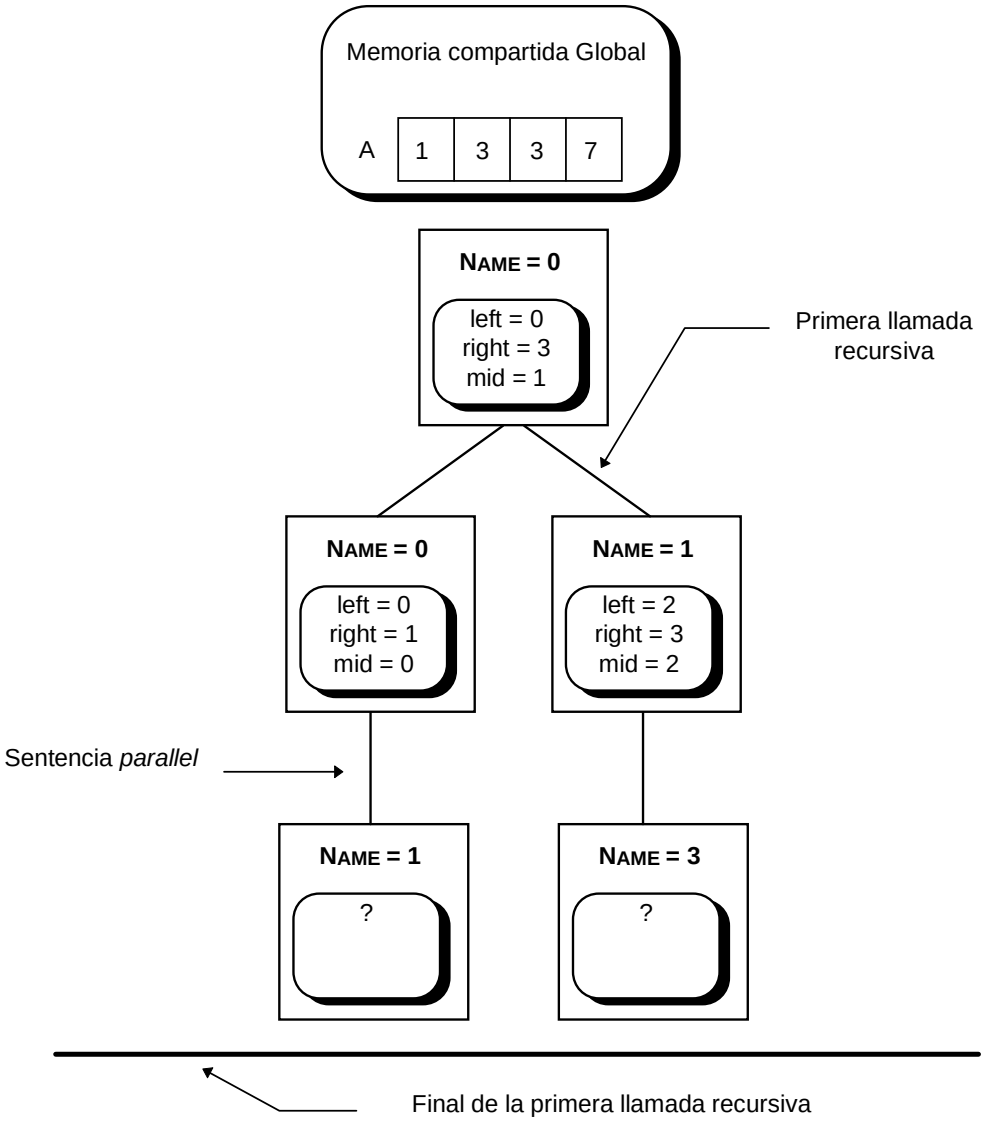
Supongamos que tenemos la siguiente declaración e inicialización del array *A*:

```
TYPE vector = ARRAY [0..3] OF INTEGER;
VAR A: SHARED vector;
...
PARALLEL 0..3 DO
   A[NAME] = NAME + 1;
```

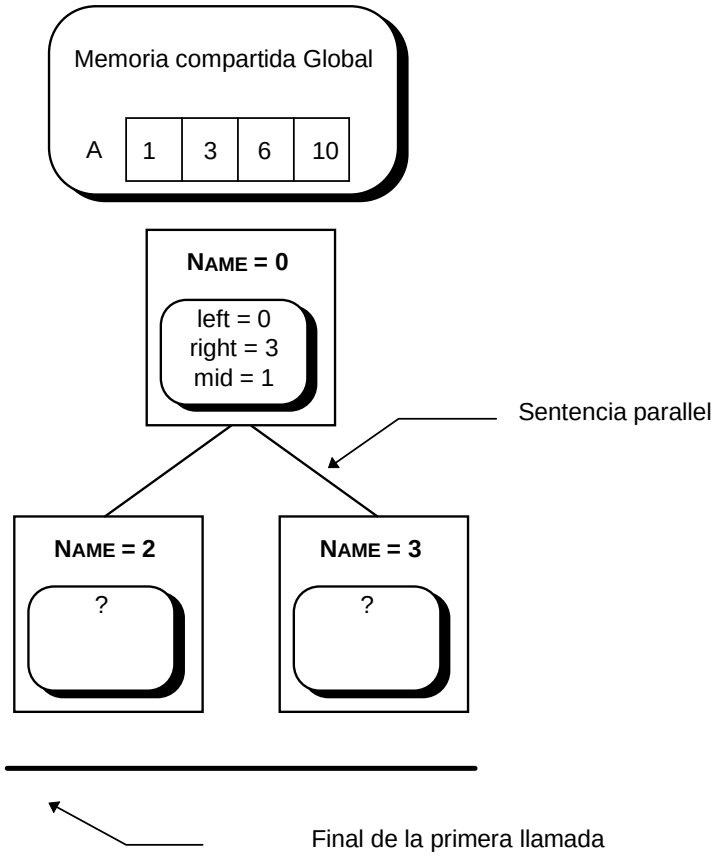
La llamada al procedimiento *prefixSum(0,3)* desde el cuerpo principal del programa activa a un procesador con nombre lógico (**NAME**) cero. La primera invocación recursiva y paralela (líneas 7 y 8) activa a dos nuevos procesadores, con nombres lógicos cero y uno, mientras que después de la segunda llamada hay activos cuatro procesadores (pues se activan dos nuevos procesadores por cada una de las llamadas paralelas anteriores).



En el momento en que finaliza la segunda llamada paralela y recursiva al procedimiento, los dos procesadores activos proceden a ejecutar la sentencia *parallel* de las línea 9 y 10. En consecuencia se activan dos nuevos procesadores con nombres lógicos 1 y 3. Estos procesadores acceden a las variables compartidas de sus antepasados en el árbol de activación para realizar el cálculo de la suma.



Se pasa nuevamente al estado en el que sólo hay un procesador activo que pasa a ejecutar la sentencia *parallel* de las líneas 9 y 10, resultando necesario activar dos nuevos procesadores que calculan el resultado final, se desactivan y dan por finalizada la llamada a *prefixSum(0,3).*



El procedimiento *prefixSum* se podría declarar usando el calificador **RELAXED** para indicar que no son necesarias sincronizaciones dentro de su cuerpo.

7. Anidamiento de Sentencias *Parallel*.

La capacidad de anidar paralelismo combinado con la potencia de la recursividad permite una formulación simple y elegante de numerosos algoritmos paralelos. Consideremos, por ejemplo, el Código 8 que resuelve de manera enumerativa el problema de la mochila 0-1. Dado un conjunto *O*

de *N* objetos con pesos *w[1],...,w[N]* y beneficios *p[1],...,p[N]* y una mochila de capacidad *C*, el problema consiste en encontrar un subconjunto de objetos *S* de *O* que aporte un beneficio *B* óptimo y quepa en la mochila.

```
 1: RELAXED PROCEDURE knapsack( i, C : INTEGER; var B : SHARED INTEGER);
 2: CONST
 3:   NO = 0 ;
 4:   YES = 1;
 5: TYPE
 6:  parameters = ARRAY [NO..YES] OF INTEGER;
 7: VAR
 8:   benefit, capacity : SHARED parameters;
 9:   fits : BOOLEAN;
10:
11: BEGIN
12:   IF  (i > 0) AND (C > 0) THEN
13:     BEGIN
14:       benefit[NO]  := B;
15:       capacity[NO] := C;
16:       fits  := (C >= w[i]);
17:       IF fits THEN
18:         BEGIN
19:           benefit[YES]  := B+p[i];
20:           capacity[YES] := C-w[i]
21:         END
22:       ELSE benefit[YES] := -INFINITY:
23:       PARALLEL NO..ord(fits) DO
24:          knapsack(i-1, capacity[NAME], benefit[NAME]);
25:       B := benefit[NO] MAX benefit[YES]
26:     END
27: END (* knapsack *);
```

Código 8. Problema de la Mochila 0/1 en ll.

El problema puede ser resuelto usando la técnica de programación divide y vencerás. Para cada objeto *i* en *O*, debemos decidir si *i* será incluido en la mochila o no. Se generan así dos subproblemas de la mochila, con conjunto de objetos *O-{i}.* Estos dos nuevos subproblemas pueden ser resueltos recursivamente en paralelo siguiendo el mismo método. Cuando resolvemos de esta manera este problema *NP completo*, el tiempo de ejecución del algoritmo analizado bajo el modelo PRAM crece linealmente con el número de objetos, O(n). Desafortunadamente el número de procesadores usados crece, en el peor caso, exponencialmente con el número de objetos.

Una forma aún más simple de codificar en ll este mismo algoritmo se encuentra en el Código 9. La declaración e inicialización de los arrays *capacity* y *benefit* necesarios en el Código 7 se logra evitar en el Código 9 mediante el uso de la sentencia de llamada paralela de las líneas 8-9 que genera dos nuevas llamadas paralelas a la rutina *knapsack* con diferentes parámetros.

```
1 : RELAXED PROCEDURE knapsack(    i, C : SHARED INTEGER;
                                   var B : SHARED INTEGER);
2 : BEGIN
3 :   IF (C > 0) AND (i > 0) THEN
4 :     IF C >= w[i] THEN
5 :       VAR Bwith, Bwithout : INTEGER;
6 :       BEGIN
7 :         Bwithout := B; Bwith := B+p[i];
8 :         PARALLEL DO
9 :           knapsack(i-1,C,Bwithout) || knapsack(i-1,C-w[i],Bwith);
10:         B := Bwith MAX Bwithout
11:       END
12:     ELSE
13:       knapsack( i-1, C, B)
14: END (* knapsack *);
```

Código 9. Propuesta Alternativa al Problema de la Mochila.

8. El Modelo de Complejidad PRSW.

En el análisis de algoritmos paralelos bajo el modelo PRAM se considera que todas las instrucciones, incluyendo las de acceso a memoria se contabilizan con costo unitario. El análisis de un algoritmo escrito en ll usando la versión 2.0 o la versión 3.0 del sistema puede hacerse fácilmente mediante el modelo que introducimos en este párrafo y que denominaremos PRSW (Parallel Read Sequential Write PRAM). Este análisis es muy parecido y tan sencillo como el análisis mediante el modelo PRAM. La complejidad PRSW de cualquier sentencia ll que no sea la sentencia de asignación se computa igual que en el modelo CRCW PRAM. La diferencia está en que la ejecución de una instrucción de asignación ll a una variable compartida, cuando es realizada en paralelo en la red de transputers, debe contabilizarse con un costo igual al número de procesadores que comparten

dicha variable y que están siendo utilizados en el momento de realizar la asignación.

Esta forma de computar el costo de una asignación puede verse como el resultado de dos hechos:

1. Toda escritura concurrente es realizada secuencialmente.
2. En el modelo PRSW se produce una ampliación del concepto de escritura concurrente. En el modelo PRAM clásico se considera que hay escritura concurrente cuando varios procesadores escriben simultáneamente en la misma dirección de memoria. En el modelo PRSW se considera que hay escritura concurrente cuando varios procesadores escriben simultáneamente en la misma variable.

Así, en el siguiente fragmento de código ll, la asignación en la variable *a* de tipo vector es considerada una escritura concurrente, aún cuando los elementos individuales *a[i]* accedidos por los *N* procesadores sean distintos. Dado que la escritura concurrente sobre el vector *a* es "secuencializada", la complejidad de hacer:

```
PARALLEL 1..N DO a[NAME] := b[NAME];
```

es O(N), que es la misma obtenida al hacer la asignación secuencialmente:

```
FOR 1 TO N DO a[COUNTER] := b[COUNTER];
```

Ejemplo 1: Suma en el Modelo PRSW.

Asumamos las declaraciones de las líneas 1-7 del Código 10 en las que *sqrtN* denota la raíz cuadrada de *N*, el código ll que le sigue almacena en la variable *result* la suma de los elementos del vector *A*.

```
 1: TYPE
 2:    vector = ARRAY [0..N-1] of INTEGER;
 3:    smallvector = ARRAY[0..sqrtN-1] of INTEGER;
 4: VAR
 5:    A : SHARED vector;
 6:    aux : SHARED smallvector;
 7:    result : INTEGER;
 8:              ......
 9: PARALLEL 0..sqrtN-1 DO
10:   VAR sum, first, last : INTEGER;
11:   RELAX BEGIN
12:    first := sqrtN*NAME;
13:    last  := first + sqrtN - 1;
14:    sum := 0;
15:    FOR first TO last DO
16:      sum := sum + A[COUNTER];
17:    aux[NAME] := sum:
18:   END (* RELAX *);
19: result := 0;
20: FOR 0 TO sqrtN-1 DO
21:   result := result + aux[COUNTER];
```

Código 10. Suma de los Elementos de un Array.

El algoritmo resuelve el problema de la agregación de N elementos mediante una operación asociativa con una complejidad PRSW O(sqrt(N)) usando sqrt(N) procesadores. En efecto, el bucle de las líneas 15-16 consume tiempo O(sqrt(N)) ya que la asignación de la línea 16 se hace en tiempo constante. Esto es así porque la variable *sum* no es compartida con otros procesadores. La asignación a la variable compartida *aux* de la línea 17 consume en el modelo PRSW tiempo O(sqrt(N)) puesto que sqrt(N) procesadores acceden a *aux*. Esto contrasta con el modelo PRAM clásico, en el que esta asignación consume tiempo unidad. Las asignaciones en las líneas 12, 13 y 14 lo son a variables privadas y, por tanto, sólo consumen tiempo unidad. Por último, el bucle secuencial final 20-21 consume tiempo O(sqrt(N)). Así la complejidad total resulta ser O(sqrt(N)) usando sqrt(N) procesadores. Puesto que el producto

Tiempo Paralelo x Número de Procesadores

es igual al tiempo del mejor algoritmo secuencial, O(N), el algoritmo utilizado es óptimo. □

Ejemplo 2: Producto de Matrices en el Modelo PRSW.

Si analizamos según el modelo PRSW el algoritmo del Código 1 para el cálculo del producto de matrices observamos que las asignaciones de las líneas 12 y 14 toman tiempo constante. La asignación a la variable compartida de la línea 15 la realizan N^2 procesadores por lo tanto tiene complejidad $O(N^2)$. Por tanto la complejidad del algoritmo es N^3. ¡La misma que la del algoritmo secuencial!. Este resultado contrasta con la complejidad O(N) obtenida haciendo un análisis según el modelo PRAM. No obstante, es posible obtener un algoritmo óptimo para el modelo PRSW con el algoritmo alternativo del Código 11.

```
 1:  PARALLEL 1..N DO
 2:    VAR sum , j: INTEGER;
 3:    RELAX
 4:      FOR 1 TO N DO
 5:        BEGIN
 6:          sum := 0; j:= COUNTER;
 7:          FOR 1 TO N DO
 8:            sum := sum+a[NAME][COUNTER]*b[COUNTER][j]
 9:          c[NAME][j] := sum
10:       END
```

Código 11. Implementación Alternativa al Producto de Matrices.

El algoritmo usa N procesadores y realiza el producto de matrices en tiempo $O(N^2)$. En efecto, el bucle 4-10 se ejecuta N veces. El tiempo invertido en cada ejecución del bucle viene dado por la suma del tiempo invertido en las asignaciones de la línea 6, el bucle interior 7-8 y en la asignación de la línea 9. Las asignaciones a las variables privadas de la línea 6 llevan tiempo constante. El bucle interior 7-8 se ejecuta N veces. La asignación de la línea 9 a la variable compartida c es realizada por N procesadores y por tanto conlleva en el modelo PRSW tiempo O(N). Por tanto, el tiempo PRSW del algoritmo es $O(N^2)$. Se cumple que el producto del tiempo paralelo $O(N^2)$ por el número de procesadores N es igual al tiempo del algoritmo secuencial $O(N^3)$.

□

Ejemplo 3: El Problema de la Mochila.

Si observamos con detenimiento el algoritmo para la resolución del problema de la mochila del Código 7, veremos que en la línea 24, el parámetro formal B, pasado por referencia, es sustituido por el parámetro actual *benefit[name].* Así, la única asignación a una variable compartida ocurre en la línea 25

B := benefit[NO] MAX benefit[YES]

La variable B es compartida a lo sumo por los dos procesadores activados en la sentencia parallel de la línea 23 y de aquí que el tiempo invertido en la ejecución de la asignación sea constante. Por tanto la complejidad PRSW del algoritmo coincide con la complejidad O(N) obtenida para el modelo PRAM.

□

Conservación del Análisis de Complejidad PRSW en la Implementación para la versión 3.0.

Dado que la implementación limita el tamaño de todo objeto declarado en ll de manera que sea posible alojarlo en la memoria de un transputer, todo acceso a memoria por un transputer lo será bien a su memoria local bien a la de sus antepasados en el árbol.

Una característica fundamental de la implementación es que cada transputer del árbol dispone de una réplica de la memoria de cada uno de sus antepasados. Así, un transputer no sólo dispone de su memoria de trabajo, sino de una réplica, que en todo momento se mantiene coherente, de la memoria de trabajo de sus ancestros. Esto significa que todo acceso a una celda de memoria para lectura se realiza en tiempo constante.

Consideremos el árbol ternario con treinta transputers distribuidos en tres niveles de la Figura 4 con 4 Mb por transputers. Asumamos que para ejecutar un programa se necesitan 132 Kb de memoria compartida. La mitad de esta memoria es accedida por todos los procesadores, mientras que la restante es compartida sólo por procesadores que pertenezcan a grupos ejecutando la misma tarea paralela. Una posible distribución es asignar 64 Kb de memoria compartida en el transputer raíz y los restantes 64 Kb en el primer nivel de transputers. Los transputers hoja no necesitan tener memoria compartida. La Tabla VI muestra las propiedades de segmentos de memoria que se establecen entre los diferentes transputers y el espacio de direcciones compartido.

NIVEL	**0**	**1**	**2**
TRANSPUTER	T0	T1, T2, T3	T4, T5, T6, T7, T8, T9, T10, T11, T12
PROPIEDAD	0 - 65.535	65.536 - 131.071	—
REPLICA	—	0 - 65.535	0 - 131.071
PRIVADA	4 Mb - 64 K	4 Mb - 132 K	4 Mb - 132 K

Tabla VI . Distribución de la Memoria Compartida en un Árbol de Tres Niveles.

Todo acceso a una celda de memoria para escritura deberá ser propagado a todos los procesadores descendientes del transputer "propietario" de dicha celda con objeto de mantener actualizadas las réplicas existentes en los mismos. Una excepción a esta regla la constituye la escritura en una variable declarada privada: dado que la variable no puede ser usada en sentencias parallel, el nuevo valor no necesita ser propagado a los descendientes del procesador que "posee" la variable.

Desde la perspectiva del análisis de complejidad, el caso peor se da cuando p procesadores virtuales escriben en p celdas de memoria situadas en el (que son propiedad del) mismo transputer T. Los nuevos p valores deben ser actualizados en todos los transputers descendientes de T. Es fácil encontrar algoritmos con los que la propagación puede realizarse sobre un árbol en tiempo de orden O(p).

En resumen tenemos los siguientes tipos de instrucciones:

- Instrucciones que tratan con operaciones de *escritura en la memoria compartida*, con un costo de implementación del mismo orden que el número de procesadores *p* que participan en el acceso al objeto.
- Instrucciones de *sincronización* que se implementan con un costo logarítmico función del número de procesadores.
- Instrucciones de *activación de procesadores*, con un costo logarítmico en el número máximo de procesadores requeridos.
- Las restantes instrucciones (operaciones de lectura, operaciones aritméticas o lógicas, operaciones de escritura en la memoria privada, etc) tienen un costo constante.

Así pues, salvo el factor logarítmico introducido por la activación de procesadores y la sincronización, la implementación realizada garantiza la conservación de la complejidad de un algoritmo PRAM analizada según el modelo PRSW.

La memoria compartida en sistemas distribuidos.

Muchos trabajos teóricos han probado la factibilidad de simular una máquina PRAM usando una red hipercúbica de procesadores. Esta parece la elección más razonable para una posible implementación de una PRAM sobre un sistema basado en redes de transputers. Para discutir las ventajas y desventajas de la implementación de una de estas simulaciones, fijemos el algoritmo descrito en [Lei91] como una representación de los trabajos teóricos. En ese algoritmo, la simulación determinista de cada acceso a la memoria en una PRAM con *N* procesadores en una máquina con topología de mariposa, toma tiempo *O(logM logN log log N). M* representa el tamaño de la memoria compartida.

El número total de pasos T_A que necesita un algoritmo PRAM *A* es la suma de tres componentes: los pasos de computación T_C, el número de accesos para escritura T_W y el número de accesos para lectura T_r:

$$T_A = T_C + T_W + T_r$$

Por lo que el número total de pasos T_A^1 requeridos por el algoritmo *A* cuando los requerimientos de memoria compartida se simulan de acuerdo con [Lei91] es:

$$T_A^1 = T_C + C \log M \log N \log\log(T_W + T_r)$$

Donde *C* denota la constante asociada al algoritmo.

Sea $q = T_r / T_W$ la razón entre el número de operaciones de lectura y escritura. Entonces T_A^1 se transforma en:

$$T_A^1 = T_C + C \log M \log N \log\log N(1+q)T_W$$

Puesto que en la implementación propuesta los requerimientos para lectura toman tiempo constante y el algoritmo de propagación para escrituras toma *C' N*, el número de pasos consumidos por el algoritmo *A* cuando se simulan las peticiones de memoria mediante "*replicación total*" será:

$$T_A^2 = T_C + C'NT_W + T_r = T_C + (C'N)T_W$$

La diferencia entre los tiempos T_A^1 y T_A^2 está marcada por los factores f_A^1 y f_A^2 que multiplican a T_w:

$$f_A^1 = C \log M \log N \log\log N(1+q)$$

y

$$f_A^2 = C'N + q$$

Una lectura detallada de [Lei91] persuadirá al lector de que la constante *C* involucrada es grande (del orden de miles) y mucho mayor que

C' (que es del orden de varias unidades). Cuando la razón entre lecturas y escrituras requeridas, *q*, es grande y el número de procesadores *n* es moderado (varios cientos), el sumando *(C log M log N log log N)q* en f_A^1 hace a este factor mayor que f_A^2. Trivialmente se sigue el siguiente lema:

Lema.- Si $q > N$ entonces $T_A^1 > T_A^2$.

Demostración:

$$f_A^2 = C'N + q < C'q + q = (C'+1)q < C \log M \log N \log\log N q < f_A^1$$

Como hemos visto, existe una clase de algoritmos PRAM donde la razón $q = T_r / T_W$ tiende a infinito cuando el tamaño del problema crece.

9. Comparando ll con FORK y fork95.

Los lenguajes ll, FORK y su dialecto fork95 se han desarrollado de forma totalmente independiente, por lo que aunque su objetivo es el mismo, presentan dos formas distintas de entender la programación paralela orientada al modelo PRAM. En esta sección se pretende comparar ambas filosofías. Ya se han introducido las sentencias de activación de procesadores en ambos lenguajes. A continuación en el epígrafe 9.1, se discute el coste de implementación de las diferentes sentencias de activación utilizando un ejemplo basado en el paradigma divide y vencerás. En el epígrafe siguiente, se presenta una implementación en ll (usando una sentencia forall clásica) del algoritmo de ordenación en fork95 que los autores definían como imposible o difícil de implementar usando sentencias for all clásicas. Por último se resumen las características más relevantes de los lenguajes cuyo estudio hemos abordado.

9.1 Coste de la Activación de Procesadores: Un Ejemplo Basado en la Técnica Divide y Vencerás

En esta sección discutiremos con más detalle el coste potencial de cada una de las sentencias paralelas que se han descrito para fork95 y para ll. Para la discusión consideremos el ejemplo genérico de algoritmo divide y vencerás del Código 12 presentado por los autores de FORK en [Hag91, Hag92]. El parámetro N del procedimiento DC describe el tamaño del problema. Los parámetros adicionales contienen los datos necesarios. Se asume además que cada paso de la recursividad divide el problema en $\sqrt{N}$ subproblemas de tamaño $\sqrt{N}$.

```
procedure DC(shared const N: integer; ...);
if trivial (N) then conquer(...)
else
  for all i in 1..sqrt(N) in parallel do
    DC(sqrt(N),...);
  combine(...);
endif;
```

Código 12. Algoritmo Divide y Vencerás Genérico.

Siguiendo las definiciones de sentencia paralela consideradas en la sección 2.4 del capítulo II, se empieza con un único procesador y sucesivamente se inicializan más procesadores puesto que cada nueva subtarea es resuelta por un procesador.

Los autores de FORK y fork95 defienden que una de sus ventajas es que la definición de la sentencia *fork* en el algoritmo DC (Código 13) hace que el grupo hoja de procesadores se subdivida sucesivamente y se distribuyan las tareas en tiempo constante, obviando el coste de inicialización de los procesadores (es decir, la activación consumiría tiempo constante). Por supuesto, se ha tenido que pagar un coste de activación de procesadores en la sentencia *start* que es necesaria para poder realizar la llamada a DC. Dicho coste será igual al logaritmo del número total, M, de procesadores necesarios. En este caso:

log(M)=log(sqrt(N)+sqrt(sqrt(N))+sqrt(sqrt(sqrt(N)))+...+1)

```
void DC(sh int N, ...);
if ( trivial(N) ) conquer(...)
else
  fork(sqrt(N)-1; @ =  /sqrt(N) ;  =  % sqrt(N))
    DC(sqrt(N),...);
  combine(...);
endif;
```

Código 13. Técnica Divide y Vencerás en fork95.

Se ha argumentado en [Kes95a] que la implementación de la semántica de la sentencia de activación de procesadores en ll conlleva un coste superior al necesario, pues cada vez que se produce una activación de procesadores se debe pagar un coste logarítmico en el número de procesadores activados. Este razonamiento es falso y está basado en la idea de una implementación directa (como la que se realizó en las versiones 2.0 y 3.0) de la semántica de la sentencia de activación propuesta en ll, pero que no es consustancial a la semántica.

Para demostrarlo, supongamos que al comienzo de la computación se cuenta con *M* procesadores físicos proporcionados por el sistema paralelo. Asociado a cada procesador existe un número de procesador físico *f*, con $f \in H = \{0,\ldots,M-1\}$. Por supuesto, la máquina paralela al ser activada inicializa los *M* procesadores de los que dispone en tiempo O(log M). Todos los procesadores replican la ejecución del programa secuencial hasta que se alcanza una sentencia paralela de la forma:

PARALLEL a..b DO instrucción(NAME)

Cada procesador dependiendo de su número *f* de procesador físico asigna el valor correspondiente a la variable número de procesador lógico **NAME** entre *a* y *b* y cambia su nombre físico. Por ejemplo, una posible política de asignación podría ser

$$\text{NAME} = a + \frac{f}{\frac{M}{b-a+1}}$$

$$f = f \bmod (b-a+1)$$

$$M = \frac{M}{(b-a+1)}$$

por lo que el tiempo invertido en la sentencia paralela es constante. Ahora los *M* procesadores en *H* han sido divididos en *r = b-a+1* conjuntos $\{H_1,\ldots,H_r\}$ Todos los procesadores dentro del conjunto H_i ienen el mismo valor de NAME, el mismo estado y replican la misma ejecución. Al llegar a la ejecución de una sentencia paralela anidada:

PARALLEL c[NAME]..d[NAME] DO instrucción(NAME)

los procesadores en el conjunto H_i se dividen en tiempo constante en *d[NAME]-c[NAME]+1* conjuntos. El procesador *f* inicializa el identificador **NAME** de manera adecuada y cambia el nombre físico. Siguiendo la política expuesta anteriormente, haría:

$$\text{NAME} = c[\text{NAME}] + \frac{f}{\frac{|H_i|}{d[\text{NAME}]-c[\text{NAME}]+1}} = c[\text{NAME}] + \frac{f}{\frac{|M|}{d[\text{NAME}]-c[\text{NAME}]+1}}$$

$$f = f \mod (d[NAME]-c[NAME]+1)$$

$$M = M \mod (d[NAME]-c[NAME]+1)$$

Existe un valor de M por cada grupo de procesadores replicados.

El razonamiento que hemos propuesto en el párrafo anterior prueba que la implementación de sentencias paralelas anidadas puede hacerse en tiempo constante.

9.2 ¿Un Algoritmo que no se puede Escribir con Sentencias for all...?

En el capítulo II, sección 2.4 se presentó la justificación que los diseñadores de FORK daban a la introducción de una semántica no natural para las sentencias paralelas (*fork*). Los autores de FORK sugieren también, la existencia de algoritmos en los que la utilización de sentencias paralelas con la semántica clásica no contribuye en absoluto a la claridad de los mismos ([Hag92] pág. 310). El único ejemplo mostrado por los autores de FORK de tal clase de algoritmos es el algoritmo de ordenación del Código 14 [Bha]. Se trata de un algoritmo que expresado en FORK o en fork95 adquiere una elegancia extraordinaria.

```
1    #include <fork.h>
2    #include <assert.h>
3    #include <io.h>
4    #define Nmax
5    sync void quicksort( sh int *);
6    pr int x, pos = 0;
7    sh int a[];
8    main(){
9      start {      /* se inicializan todos los procesadores disponibles */
10      sh int numElem;
11      x = a[];
12      quicksort( &numElemt );
13      a[pos-1] = x;
14      } /* start */
15   }/* main */
16   sync void quicksort( sh int *numElement) {
17     sh int pivot, numLeft = 0, numRight = 0;
18     pr int left, right;
19     pivot = x;
20     left =  (x < pivot);
21     right = (x > pivot);
22     if (x != pivot)
23        if (left)  quicksort( &numLeft );
24        else     quicksort( &numRight );
25     if (x == pivot)  pos = numLeft + 1;
26     if (right) pos += 1 + numRight;
27     *numElement = numLeft + numRight + 1;
28   } /* quicksort */
```

Código 14. Algoritmo de Ordenación [Bha] en fork95.

En el acceso concurrente para escritura de la línea 19, todos los procesadores activos intentan escribir su valor a ordenar, *x,* en la variable compartida *pivot*. Uno de los procesadores gana, se divide el vector por ese punto y se procede a ordenar los subconjuntos obtenidos mediante llamadas recursivas al mismo procedimiento (líneas 22 a 24). A continuación en las líneas 25 y 26 cada procesador calcula la posición relativa de su elemento del vector a ordenar.

Aunque los autores no lo señalen, nótese que para que el algoritmo del Código 14 funcione correctamente se debe asumir que todos los elementos del vector a ordenar son distintos entre sí. La complejidad de este algoritmo es O(n), aún asumiendo que las escrituras concurrentes pudieran efectuarse en tiempo constante; si se cargan, como es habitual, con un coste logarítmico, el algoritmo adquiere la misma complejidad que el algoritmo secuencial. Esta es una complejidad pobre si se la compara con otros algoritmos de ordenación para el modelo PRAM [JáJ92] como el que aparece en el Código 16 codificado en lenguaje ll.

En el Código 15 proponemos una versión del algoritmo de ordenación [Bha] en lenguaje ll en la que se hace uso de sentencias paralelas del tipo *for all...* . Aunque la escritura en ll no resulta tan elegante como en fork95, el código prueba la factibilidad de escribir el algoritmo usando sentencias de activación con la semántica clásica.

Es posible expresar en ll otros algoritmos de ordenación igualmente elegantes y más eficientes [JáJ92], como la paralelización del *quicksort* que se muestra en el Código 16 [Hid93]. En este algoritmo, *S* almacena el vector a ordenar. De la misma forma que en el *quicksort* secuencial el vector se divide en dos segmentos, el izquierdo conteniendo los elementos menores que el *pivot* y el derecho conteniendo los elementos mayores. Estos dos intervalos son ordenados en paralelo y recursivamente en la línea 36. El proceso de división también es paralelizado (*partition*). Los procesadores cuentan el número de elementos menores que el pivote utilizando una

versión paralela de la suma de prefijos (*PrefixSum*). Esta información es suficiente para particionar el segmento inicial.

```
PROGRAM QuickSortParalelo;
CONST
  MaxN  = ...;
  MaxNmenos1 =... ;
  Max2N =... ;
TYPE
  VectInt  = ARRAY [0..MaxNmenos1] OF INTEGER;
  VectLong = ARRAY [1..Max2N] OF INTEGER;
VAR
    A         : SHARED VectInt;          (* Array a ordenar              *)
    ne,                                  (* Numero de Elementos ordenados *)
    P         : SHARED  VectLong;        (* Guarda los pivots            *)
    n         : INTEGER;                 (* Numero de Elementos del array *)

SHARED PROCEDURE sort(VAR pos: INTEGER; x : INTEGER; index : INTEGER);
VAR
  nl, nr, left, right, pivot : INTEGER;
  lefts, rights : BOOLEAN;
BEGIN
  left         := 2*index;  right      := left + 1;
  ne[left]     := 0;        ne[right] := 0;
  P[index]  := x;           pivot      := P[index];
  lefts  := x < pivot;      rights     := x > pivot;
  IF (x <> pivot) THEN
     IF(lefts) DO
       sort(pos, x, left)
     ELSE
       sort(pos, x, right);
  nl := ne[left];  nr := ne[right];
  IF (x = pivot) THEN pos := nl + 1;
  IF (rights) THEN  pos := pos + 1 + nl;
  ne[index] := nl + nr + 1;
END; (* sort *)

BEGIN (* main *)
    PARALLEL 0..n-1 DO
    VAR pos, x: SHARED INTEGER;
    BEGIN
      pos := 0;
      x := A[NAME];
      ne[1] := 0;
      sort(pos, x, 1);
      A[pos-1] := A[NAME];
    END; (*paralle do*)
END. (*main*)
```

Código 15. Algoritmo de Ordenación [Bha] en ll.

```
1   SHARED PROCEDURE partition( left, right : SHARED INTEGER ;
2                               VAR PivotPos : SHARED INTEGER);
3    VAR pivot : SHARED INTEGER;
4    SHARED PROCEDURE PrefixSum( i, j : SHARED INTEGER);
5    (* algoritmo paralelo para la suma de prefijos en el array sum [JaJa]*)

6   BEGIN
7    pivot := S[pivotPos];
8    PARALLEL left..right DO
9     sum[name] := ord(S[name]<pivot);
10   PrefixSum( left, right);
11   PARALLEL left..right DO
12   VAR position : INTEGER;
13    BEGIN
14     IF name <> pivotPos THEN
15      BEGIN
16       IF S[name] < pivot THEN
17        position := sum[name] + left - 1
18       ELSE (* (S[name] >= pivot) AND (name <> pivotPos) *)
19        position := name + sum[name] + ord(name < pivotPos)
20      END
21     ELSE (* name = pivotPos *)
22      BEGIN (* colocar el pivot *)
23       pivotPos := sum[right] + left; (* nueva posicion del pivot *)
24       position := pivotPos
25      END;
26     S[position] := S[name]    (* actualizar *)
27    END  (* PARALLEL *)
28  END; (*  Partition *)

29  SHARED PROCEDURE sort( left, right : SHARED INTEGER );
30  VAR pivotPos : SHARED INTEGER;
31  BEGIN
32   IF left < right THEN
33    BEGIN
34     pivotPos := (left + right ) DIV 2;
35     partition( left, right , pivotPos );
36     PARALLEL DO
37         sort( left, pivotPos - 1 ) || sort( pivotPos + 1, right );
38    END; (* IF left < right *)
39  END; (* sort *)

40  ... sort( 1, n ); ....
```

Código 16. Algoritmo de Ordenación [JáJ92] en ll.

FORK es un lenguaje con una extraordinaria capacidad para la expresión de algoritmos PRAM. FORK ha sido desarrollado dentro de un vasto proyecto que implica no sólo al lenguaje sino también la creación de una nueva máquina paralela que en su última versión integra 128 procesadores realizando la simulación de Ranade [Ran89a, Ran89b]. La

poda introducida en fork95, aunque drástica ha beneficiado al lenguaje, haciendo más factible la realización de un compilador. La realización de un compilador de fork95 que produzca código para una auténtica máquina paralela permanece a la espera de la construcción de la SBRAM. Actualmente, el lenguaje dispone de un compilador para estaciones de trabajo [Kes95a]. Aunque el desarrollo del lenguaje ll se ha llevado a cabo de forma totalmente independiente a la de fork95, las similitudes entre ambos lenguajes son mayores que las diferencias. La Tabla VII resume las características más relevantes que se pueden encontrar en los tres lenguajes de programación orientados al modelo PRAM cuyo estudio hemos abordado.

	fork95	**ll**	**FORK**
Virtualización	No	SI	SI
Sincronización PRAM	Débil	SI	SI
Coste de activación	Constante	Constante	Constante
Recursividad y Paralelismo	SI	SI	SI
Sincronía Gruesa (Relajación de la Sincronización PRAM)	SI	SI	NO
Activación de un Número Privado de Procesadores	NO	SI	NO
Implementación	SI(SUN)	SI (Pcs, Transputers)	NO

Tabla VII. Principales Características de FORK, fork95 y ll.

Conclusiones y Trabajos Futuros

El lenguaje y los compiladores de ll han sido desarrollados en la Universidad de La Laguna con la financiación del Gobierno de Canarias. Actualmente se dispone de compiladores de ll para máquinas compatibles IBM-PC y redes de transputers. Como ya hemos comentado, el lenguaje ll ha demostrado su sencillez a la hora de presentar de forma elegante y estructurada una amplia variedad de algoritmos. Si bien en sus inicios surgió como un lenguaje para algoritmos orientados al modelo PRAM, ha demostrado su validez en la codificación de algoritmos tipo divide y vencerás, greedy, programación dinámica, etc.. Este particular puede corroborarse a la vista de los distintos proyectos que se han realizado en el Centro Superior de Informática de la Universidad de La Laguna utilizando el sistema ll como herramienta de desarrollo. Aunque su desarrollo ha tenido

lugar de forma totalmente independiente del de fork95, las similitudes entre ambos lenguajes son mayores que las diferencias.

1. Eficiencia del Sistema ll.

La crítica que se suele hacer a los lenguajes orientados al modelo PRAM, como son ll y fork95, es la cuestión de si es factible una traducción eficiente a máquinas paralelas "reales". La respuesta a esta pregunta en el caso de compiladores de fork95 está a la espera de la construcción de la máquina SBPRAM [Abo90, Kel94]. Aún cuando la eficiencia de la implementación actual del compilador ll es pobre para una clase de algoritmos suficientemente general, utilizando un compilador para transputers de un subconjunto reducido de ll hemos obtenido para el producto de matrices aceleraciones comparables a las obtenidas programando directamente en occam un algoritmo similar. El algoritmo en occam fue escrito para una topología de árbol y se presenta en el apéndice B. En el algoritmo occam las filas de A y columnas de B fueron convenientemente replicadas en los procesadores y cada transputer computa una submatriz cuadrada de la matriz resultado C que es luego enviada al procesador raíz. La Tabla I muestra las aceleraciones obtenidas para sistemas con diferente número de procesadores por los algoritmos programados en los lenguajes ll y occam para el producto de matrices 300x300.

transputers	3	7	15
Occam	2.7	5.8	10.7
ll	2.4	4.7	8.5

Tabla I. Aceleraciones para matrices de tamaño 300 x 300

2. Posibles Mejoras al Lenguaje ll.

El diseño inicial del lenguaje ll presenta dificultades en tres aspectos:

- Virtualización
- Grupos Síncronos
- Distribución Automática de la Carga

Virtualización.

El coste de depositar la virtualización de los procesadores en el software resulta excesivo para el lenguaje ll. Se gasta excesivo tiempo en cambios de contexto y en conmutación de procesos.

Grupos Síncronos.

Se observa también en el lenguaje, la carencia de una sentencia condicional que permita crear grupos síncronos que sean asíncronos entre sí. Este tipo de sentencias condicionales dotarían al lenguaje de una semántica más "paralela" como la que proporcionan Modula 2* o fork95.

Distribución Automática de la Carga.

Depositar la distribución de la carga de trabajo en el sistema puede ser agradable para el programador, pero en algoritmos con patrones irregulares, como el algoritmo de ordenación del Código 1 (sección 5), en el que una elección errónea del pivote (*pivot*) puede conducir a particiones de tamaños muy diferentes, los resultados son desastrosos si se aplica un equilibrado de la carga equitativo como el que sigue el sistema ll.

Puesto que el conocimiento que tiene un programador sobre el algoritmo que intenta implementar es más profundo que el que pueda llegar

a extraer cualquier compilador, se hace necesario proporcionar al usuario recursos que le permitan indicar al sistema cual es la distribución de carga adecuada.

3. Posibles Mejoras de la Implementación del Sistema ll.

Otros inconvenientes que presenta el sistema ll, se refieren no tanto al diseño del lenguaje, sino a las decisiones tomadas durante la implementación de los sucesivos compiladores. Específicamente se trata de:

- La Elección de la Máquina Objeto y el Lenguaje Objeto.
- Coste de implementación de las Sentencias *parallel*.

La Máquina Objeto y el Lenguaje Objeto.

La elección de una red de transputers y del lenguaje occam como máquina y lenguaje objetos, respectivamente, para traducir un lenguaje orientado al modelo PRAM supuso probablemente elegir una de las plataformas más difíciles e inadecuadas. La decisión estuvo forzada en su momento por consideraciones económicas y de disponibilidad (desgraciadamente la situación en la Universidad de La Laguna no ha cambiado sustancialmente, aunque actualmente la mayor disponibilidad de uso de máquinas remotas introduce un nuevo y esperanzador elemento a considerar).

El coste de las sincronizaciones por barrera en la red de transputers es costoso. Desgraciadamente estas sincronizaciones en los algoritmos PRAM ocurren con frecuencia. Se necesita una máquina objeto que sea más eficiente en las sincronizaciones por barrera. Mejor aún para la

traducción del lenguaje ll sería disponer de una máquina de memoria compartida.

La falta de recursividad en el lenguaje occam y la ausencia de punteros hicieron la tarea de traducción un trabajo más arduo del necesario. Mientras el sistema ll fue interpretado, occam supuso una ventaja, puesto que proporciona una seguridad en la programación paralela que pocos o quizá ningún otro lenguaje paralelo pueda ofrecer. Seguramente Inmos C hubiese sido una mejor elección para la version compilada, pero por razones de “reciclaje”, en el momento de pasar a la compilación del código intermedio optamos por occam.

Coste de implementación de las Sentencias parallel.

El coste de la traducción de las sentencias de activación de procesadores (sentencias *parallel* y sentencias de llamada a procedimiento paralela) es logarítmico en la implantación actual. Como se ha discutido en la sección 9.1, del capítulo III, este coste es innecesario y usando la estrategia de replicación del cómputo puede realizarse en tiempo constante.

En un intento de simplificar la programación de la primera versión paralela (versión 2.0) se realizó una implementación en la que los procesadores inactivos permanecen ociosos a la espera de activación por parte de sus ancestros en la jerarquía, en lugar de replicar el cómputo de estos. Aunque esta decisión facilitó la elaboración del sistema, provocó una ejecución más lenta de las sentencias de activación de procesadores.

A continuación describimos los principales cambios que introduce la nueva versión (4.0) del sistema ll. Por razones de coherencia en la exposición presentamos primero las respuestas que pretendemos dar en la nueva versión de ll a las deficiencias en la implementación y luego las respuestas a las deficiencias observadas en el diseño del lenguaje.

4. Propuesta de Implementación del Sistema II.

Ejecución Replicada.

Como se demostró en el apartado 9.1 del capítulo III, la activación de procesadores en ll no tiene porqué consumir más que tiempo constante usando el algoritmo de replicación descrito en esa sección. Los procesadores replicantes en un mismo grupo ejecutan el mismo código. Todas sus variables así como el nombre lógico y el contador de programa son idénticos. Sólo el nombre físico es diferente en cada uno de ellos. Uno de los procesadores actúa como "maestro" y tiene derechos de escritura en memoria compartida, los demás se inhiben de realizarlas. El número de procesadores replicantes de una computación dada está disponible para el programador a través de la variable **NUMPROC**.

La Máquina Objeto y el Lenguaje Objeto.

Actualmente estamos en condiciones de acceder a una máquina de memoria compartida que ofrece un buen rendimiento en sincronización por barrera y que dispone de compiladores FORTRAN 90 y C que paralelizan automáticamente. Se trata de una *Silicon Power Challenge XL* con seis procesadores MIPS superescalares que ha sido adquirida recientemente por el Instituto de Astrofísica de Canarias (IAC). Una máquina con estas características está también disponible en el Centro Europeo de Paralelismo de Barcelona (CEPBA). Se trata de un procesador multisimétrico capaz de soportar un máximo de 36 procesadores y que proporciona más de 13.5 GFLOPS de rendimiento pico. El sistema utiliza un bus de tipo POWERpath-2 y varios niveles de memoria caché con un protocolo de coherencia de tipo SNOOPY. Sin duda este sistema reune mejores condiciones para la implementación de un lenguaje como ll, que la red de transputers utilizada. Se da el caso de que una implementación

sobre una máquina como esta puede producir una doble paralelización: la que proporciona ll y la que produce el compilador del sistema del lenguaje elegido como objeto.

5. Modificaciones al Lenguaje ll.

Supresión de la Virtualización.

Uno de los cambios a introducir en el sistema ll consiste en dejar que la virtualización de procesadores sea soportada por el hardware en lugar de dejarla a cargo del sofware. Aunque la virtualización de procesadores contribuye a la portabilidad de los programas, tiene un coste prohibitivo cuando no es soportada de forma conveniente por el hardware: Para una implementación eficiente de la virtualización se necesitan máquinas extraordinariamente rápidas en la realización de cambios de contexto y conmutación entre procesos.

La variable de sólo lectura **NUMPROC** indicará, el número de procesadores físicos disponibles. Esta variable puede ser consultada por cualquier procesador, y esto a nivel de implementación se traduce en que se conoce el cardinal del grupo de procesadores replicantes al que un procesador pertenece.

Al comienzo de la computación **NUMPROC** es igual al número de procesadores proporcionados por el sistema. En un instante dado de la computación existen tantas variables **NUMPROC** como grupos de replicantes. Así en el Código 1 la evaluación de la condición

```
IF (NUMPROC = 1) THEN ...
```

se refiere al número de procesadores replicantes en el grupo del procesador ejecutante.

```
RELAXED PROCEDURE sort( left, right : SHARED INTEGER );
  VAR pivotPos : SHARED INTEGER;
  BEGIN
    IF (NUMPROC = 1) THEN
      seqSort(left, right)
    ELSE
    IF left < right THEN
    BEGIN
      select(pivotPos);
      partition( left, right , pivotPos );
      sort( left, nright ) || sort( nleft, right );
    END; (* IF left < right *)
  END; (* sort *)
```

Código 1. Algoritmo de Ordenación.

Grupos Síncronos.

Aunque una solución parcial al problema de los grupos síncronos en sentencias condicionales viene dada por la presencia del calificador **CALM**, la solución general a este problema se articula mediante la introducción de una sentencia condicional con la misma semántica que Modula2* y FORK. Esta sentencia se denominá *where*. Su sintaxis es la siguiente:

WHERE <B> **DO**
<Sentencia1>
ELSE
<Sentencia2>

Los procesadores que ejecutan una sentencia *where* evalúan la expresión booleana <B>. Aquellos para los que <B> es cierta ejecutan <Sentencia1> síncronamente, mientras que aquellos para los que es falsa ejecutan <Sentencia2> síncronamente. Los dos grupos de procesadores que se forman trabajan asíncronamente entre sí.

Al igual que las sentencias calificadas con **RELAX** y **CALM**, la sentencia *where* supone una extensión de la concepción SIMD original de ll a un modelo de programación SPMD. Estas tres sentencias suponen que cada grupo de replicantes de la PRAM tiene su propio contador de programa y su

propio control. De aquí la necesidad de introducir dos nuevas instrucciones en el juego de instrucciones inicial del código intermedio. Una sentencia *where* se traduce siguiendo el Esquema 1.

```
                Código de <B>
                SYNC                        -- Sincronización.
                PARTITION else
                Código de <Sentencia1>
                GOTO endwhere
else:
                Código de <Sentencia2>
endwhere:
                JOIN
```

Esquema 1. Traducción de una Sentencia Where.

La instrucción **PARTITION** *label* provoca que el procesador salte a la etiqueta si su registro de Control (esto es, el valor de C[**NAME**]) es cero y que se ejecute la siguiente instrucción en caso contrario. El registro de Control se almacena en la pila y se divide en dos subregistros correspondientes respectivamente a los procesadores que han evaluado a cierto y a falso la expresión booleana.

Las instrucciones relacionadas con la sincronización y con el registro de Control afectan a estos subregistros de Control a partir de este momento. La instrucción **JOIN** es la inversa de la instrucción **PARTITION**. La misión de **JOIN** consiste en, utilizando la información almacenada en la pila sincronizar los dos grupos y formar nuevamente un único grupo con un sólo registro de Control.

Distribución de la Carga.

En la nueva versión el programador de ll podrá controlar la distribución de la carga de trabajo entre los diferentes procesadores del grupo de replicantes ejecutando una sentencia *parallel* o una sentencia paralela de llamada a subprograma. Para ello, se hacen temporalmente accesibles al programador para lectura y escritura no sólo el nombre lógico

NAME, sino también el número de procesadores disponibles por el grupo de replicantes **NUMPROC** y el nombre físico del procesador, representado por la variable **PROCESSOR**.

Se le permite al programador sustituir la distribución automática proporcionada por el sistema ll, descrita en la sección 5.2 del capítulo II, por la suya propia, modificando de manera coherente los valores de **NAME**, **NUMPROC** y **PROCESSOR**.

La sintaxis general de una sentencia *parallel* en ll 4.0 es la siguiente:

PARALLEL <Distribución> **DO**
<Sentencia>

donde la variable sintáctica <Distribución> se define por:

<Distribución>	::=	<Distribución Automática>
	|	<Distribución Manual>
<Distribución Automática>	::=	<Expresión1>**..**<Expresión2>
<Distribución Manual>	::=	<Semibloque de Distribución>
	|	<Procedimiento de Distribución>

En general una <Distribución Manual> tiene la misma forma que un procedimiento o que un semibloque de ll. En ese bloque el programador debe definir las tres variables **NAME** (nombre lógico), **PROCESSOR** (nombre físico) y **NUMPROC** (número de procesadores disponibles por el grupo de replicantes a los que pertenece **PROCESSOR**). La variable **PROCESSOR** sólo se puede utilizar dentro del cuerpo de una <Distribución>. En el ámbito de una <Distribución> no se pueden realizar escrituras a variables que no sean locales al cuerpo de la distribución. Una distribución no puede llamar a otras rutinas si no han sido declaradas del tipo **DISTRIBUTION**. Estas restricciones se introducen para conseguir que la conducta de los

procesadores dentro del mismo grupo de replicación siga siendo la misma. Esto es, que todos los procesadores dentro del mismo grupo de replicación tengan los mismos valores en todas sus variables, excepto la variable **PROCESSOR**. Esta conducta podría verse alterada si un procesador utiliza la variable **PROCESSOR** para hacer un cambio en una variable externa a la distribución. Las distribuciones son procedimientos relajados, esto es los procesadores, mientras lo ejecutan no participan en sincronizaciones.

El ejemplo del Código 2 muestra el uso de una distribución para el algoritmo del Código 1 en el que el número de procesadores asignado a cada llamada es proporcional al tamaño del intervalo a ordenar. El Código 3 muestra la distribución utilizada.

```
SHARED PROCEDURE sort( left, right : SHARED INTEGER );
VAR pivotPos : SHARED INTEGER;
BEGIN
    IF (NUMPROC = 1) THEN
       seqSort(left, right)
    ELSE  IF left < right THEN
       BEGIN
          pivotPos := (left + right ) DIV 2;
          partition( left, right , pivotPos );
          PARALLEL proportional(left, right, pivotPos) DO
              sort(left,pivotPos-1) || sort(pivotPos+1,right);
       END; (* IF left < right *)
END; (* sort *)
```

Código 2. Una Distribución Manual sobre el Código 1.

El lenguaje ll y el lenguaje FORK proponían en 1990 un modelo de programación basado en memoria compartida, sincronización por barrera y posibilidad de anidamiento de sentencias paralelas. Aún cuando ninguno de los dos lenguajes ha tenido eco en la comunidad de la supercomputación, no deja de ser curioso, que el que indudablemente se presenta como lenguaje de la supercomputación para el futuro, el lenguaje que recoge las mayores inversiones de capital y recursos humanos en el campo de la supercomputación, HPF[Lov93], sea un lenguaje que realiza las dos primeras propuestas. Actualmente es común escuchar en los congresos de la especialidad, que probablemente el mayor defecto de HPF es que sus

implementaciones actuales no son eficientes sobre datos irregulares. También conocemos la existencia de experiencias [Ble94, Har96] que parecen demostrar que la tercera propuesta de usar anidamiento de sentencias paralelas puede contribuir a resolver este problema y ésta no ha sido considerada en HPF).

```
(* Divide el conjunto de procesadores disponibles en dos grupos con numero      *)
(* de procesadores proporcional al tamaño de los respectivos intervalos         *)
DISTRIBUTION proportional(left, right, pivotPos : INTEGER);
VAR
  nleft,                  (* Numero de elementos en el subintervalo izquierdo *)
  npl   : INTEGER;    (*Numero de procesadores asignados para el subintervalo izquierdo*)
BEGIN
  nleft := pivotPos-left;
  npl := (NUMPROC * nleft) / (right - left);
  IF npl = 0 THEN            (* Para garantizar que al menos uno trabaja en el subintervalo *)
                             (* izquierdo*)
     npl := 1
  ELSE                       (* Para garantizar que al menos uno trabaja en el  *)
                             (*  subintervalo derecho *)
     IF npl = NUMPROC THEN
        npl := NUMPROC - 1;
  IF (PROCESSOR < npl) THEN   (* hacer que los menores que npl  trabjen en ordenar  *)
     BEGIN                          (* el subintervalo izquierdo                    *)
                                (*Se asume que siempre los valores de PROCESSOR estan *)
                                (* en el  rango 0..numproc-1. *)
       NAME := 0;
       NUMPROC := nl;
       PROCESSOR := PROCESSOR MOD NUMPROC;
     END
  ELSE
     BEGIN          (* hacer que los restantes trabajen en ordenar el subintervalo derecho *)
       NAME := 1;
       NUMPROC := NUMPROC - nl;
       PROCESSOR := PROCESSOR MOD NUMPROC;
     END;
END; (*proportional*)
```

Código 3. Un Ejemplo de Distribución.

Apéndice A

Gramática del lenguaje II (3.0)

<Programa> ::= **PROGRAM** <Identificador de Programa> **";"** <Bloque> **"."**

<Bloque> ::= [<Parte de Definición de Constantes>]
[<Parte de Definición de Tipos>]
[<Parte de Declaración de Variables>]
{<Definición de Procedimientos>}
<Sentencia Compuesta>

<Parte de Definición de Constantes> ::=
CONST <Definición de Constante> {<Definición de Constante>}

<Definición de Constante> ::= <Identificador de Constante> **"="**<Constante>**";"**

<Constante> ::= <Número> | <Identificador de Constante>

<Parte de Definición de Tipos> ::=

TYPE <Definición de Tipos> {<Definición de Tipos>}

<Definición de Tipo> ::= <Identificador de Tipo> **"="** <Nuevo Tipo> **";"**

<Nuevo Tipo> ::= <Nuevo Tipo Subrango>

| <Nuevo Tipo Conjunto>

| <Nuevo Tipo Array>

| <Nuevo Tipo Registro>

<Nuevo Tipo Subrango> ::= <Constante>**".."**<Constante>

<Nuevo Tipo Conjunto> ::= **SET OF** <Identificador de Tipo>

<Nuevo Tipo Array> ::= **ARRAY [**<Rango de Índices>**] OF** <Identificador de Tipo>

<Rango de Índices> ::= <Identificador Tipo Subrango>

| <Constante>**".."**<Constante>

<Nuevo Tipo Registro> ::= **RECORD** <Lista de Campos> **END**

<Lista de Campos> ::= <Sección de Registro> { **";"** <Sección de Registro>}

<Sección de Registro> ::= <Identificador de Campo>

{**","**<Identificador de Campo>} **":"** <Identificador de Tipo>

<Parte de Declaración de Variables> ::=

VAR <Declaración de Variables> {<Declaración de Variables>}

<Declaración de Variables> ::= <Grupo de Variables> **";"**

<Grupo de Variables> ::= <Identificador de Variable>

{**","** <Identificador de Variable>} **":"** [**SHARED**] <Identificador de Tipo>

<Definición de Procedimientos> ::=

[**SHARED** | **RELAXED**] **PROCEDURE** <Identificador> <Bloque de Procedimiento> **";"**

<Bloque de Procedimiento> ::=

[**"("**<Lista de Parámetros Formales>**")"**] **";"** <Bloque>

<Lista de Parámetros Formales> ::=

<Definición de Parámetro> { **";"** <Definición de Parámetro>}

<Definición de Parámetro> ::= [**VAR**] <Grupo de Variables>

<Sentencia> ::= [**RELAX** | **CALM**] <Sentencia sin Relajación>

<Sentencia sin Relajación> ::= <Sentencia de Activación Paralela>

| <Sentencia de Flujo de Control>

<Sentencia de Activación Paralela> ::= <Sentencia Parallel>

| <Sentencia de LLamada Paralela>

<Sentencia de Flujo de Control> ::=<Sentencia de Asignación>

| <Sentencia de Procedimiento>

| <Sentencia If>

| <Sentencia While>

| <Sentencia Repeat>

| <Sentencia For>

| <Sentencia de Semibloque>

| <Sentencia Compuesta>

| ∈

<Sentencia Parallel> ::= **PARALLEL** <Expresión> **".."** <Expresión>

[**THRESHOLD** <Constante>] **DO** <Sentencia>

<Sentencia de LLamada Paralela> ::=

PARALLEL DO <Sentencia de Procedimiento> **"||"**

<Sentencia de Procedimiento> { **"||"** <Sentencia de Procedimiento>}

<Sentencia de Asignación> ::= <Referencia a Variable> **":="** <Expresión>

<Sentencia de Procedimiento> ::=

<Identificador de Procedimiento> [**"("** <Lista de Parámetros Actuales> **")"**]

<Lista de Parámetros Actuales> ::=

<Parámetro Actual> { **","** <Parámetro Actual>}

<Parámetro Actual> ::= <Expresión> | <Referencia a Variable>

<Sentencia de Semibloque> ::= <Parte de Definición de Variables>

<Sentencia Compuesta>

<Sentencia If> ::= **IF** <Expresión> **THEN** <Sentencia> [**ELSE** <Sentencia>]

<Sentencia While> ::= **WHILE** <Expresión> **DO** <Sentencia>

<Sentencia Repeat> ::=

REPEAT <Sentencia> { **";"** <Sentencia>} **UNTIL** <Expresión>

<Sentencia For> ::= **FOR** <Expresión> **TO** <Expresión> **DO** <Sentencia>

<Sentencia Compuesta> ::= **BEGIN** <Sentencia> { **";"** <Sentencia> } **END**

<Expresión> ::= <Expresión Simple>

[<Operador Relacional> <Expresión Simple>]

<Operador Relacional> ::= **"<="** | **"="** | **">"** | **"<="** | **"<>"** | **">="** | **IN**

<Expresión Simple> ::= [<Operador de Signo> | <Potencia de Dos>] <Término>

{<Operador Aditivo> <Término>}

<Operador de Signo> ::= **"+"** | **"-"**

<Potencia de Dos> ::= **"&"**

<Operador Aditivo> ::= **“+”** | **“-”** | **OR**

<Término> ::= <Factor> [<Operador Multiplicativo> <Factor>]

<Operador multiplicativo> ::= **“*”** | **DIV** | **MOD** | **AND** | **SHL** | **SHR** | **XOR** | **MAX** | **MIN**

<Factor> ::= <Constante> | <Referencia a Variable> | **“(”** <Expresión> **“)”**

| **NOT** <factor> | <Constructor de Conjunto> | **RANDOM**

<Constructor de Conjunto> ::=

<Idenficador de Conjunto> **“[”** <Miembro de Grupo>

{**“,”**<Miembro de Grupo>} **“]”**

<Miembro de Grupo> ::= <Expresión>

| <Expresión> **“..”** <Expresión>

<Referencia a Variable> ::= <Identificador de Variable> {<Selector>}

<Selector> ::= <Selector de Índice> | <Selector de Campos>

<Selector de Índices> ::= **“[”** <Expresión> **“]”**

<Selector de Campos> ::= **“. ”** <Identificador de Campo>

Apéndice B

Código Occam Correspondiente a un Producto de Matrices con Replicación Total de los Datos

```
#INCLUDE "matrix.inc"

PROC node(VAL INT name, VAL INT numproc, CHAN OF COMM from.father, to.father, from.left, to.left, from.right, to.right)
  ... Variable declaration
```

```
PROC receive(CHAN OF COMM from.father, to.left, to.right)
 ... Variable declaration

 IF
  NOT IS.LEAF
        SEQ
         counter := 0
         WHILE (counter < MAX.DIM)  -- Reception of matrix A
          SEQ
           from.father ? CASE
                 row; row.num; no.of.elements :: [A[row.num] FROM 0 FOR no.of.elements]
                  SEQ
                   counter := counter + 1
                   PAR
                    to.left  !  row; row.num; no.of.elements :: [A[row.num] FROM 0 FOR no.of.elements]
                    to.right !  row; row.num; no.of.elements :: [A[row.num] FROM 0 FOR no.of.elements]

         counter := 0
         WHILE (counter < MAX.DIM)
          SEQ
           from.father ? CASE     -- Reception of matrix B
                 row; row.num; no.of.elements :: [B[row.num] FROM 0 FOR no.of.elements]
                  SEQ
                   counter := counter + 1
                   PAR
                    to.left  !  row; row.num; no.of.elements :: [B[row.num] FROM 0 FOR no.of.elements]
                    to.right !  row; row.num; no.of.elements :: [B[row.num] FROM 0 FOR no.of.elements]
  IS.LEAF
        SEQ
         counter := 0
         WHILE (counter < MAX.DIM)
          SEQ
           from.father ? CASE     -- Reception of matrix A
```

```
                row; row.num; no.of.elements :: [A[row.num] FROM 0 FOR no.of.elements]
                  SEQ
                    counter := counter + 1


          counter := 0
          WHILE (counter < MAX.DIM)
            SEQ
              from.father ? CASE     -- Reception of matrix B
                row; row.num; no.of.elements :: [B[row.num] FROM 0 FOR no.of.elements]
                  SEQ
                    counter := counter + 1
:
```

```
PROC multiply()
  INT sum :

  SEQ i = (name * RATIO) FOR RATIO
    SEQ j = 0 FOR MAX.DIM
        SEQ
          sum := 0
          SEQ k = 0 FOR MAX.DIM
            sum := sum + (A[i][k] * B[k][j])
          C[i][j] := sum
:
```

```
PROC send(CHAN OF COMM to.father, from.left, from.right)
  ... Variable declaration

  IF
    IS.ROOT
        SEQ
          SEQ i = (name * RATIO) FOR RATIO
            to.father ! row; i; MAX.DIM :: [C[i] FROM 0 FOR MAX.DIM]
          counter := 0
          WHILE (counter < (RATIO * (numproc - 1)))
            ALT
              from.left ? CASE
                    row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                      SEQ
                        to.father ! row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                        counter := counter + 1

              from.right ? CASE
                    row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                      SEQ
                        to.father ! row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                        counter := counter + 1

          to.father ! end.of.program
    NOT IS.LEAF
        SEQ
          SEQ i = (name * RATIO) FOR RATIO
            to.father ! row; i; MAX.DIM :: [C[i] FROM 0 FOR MAX.DIM]
          WHILE TRUE
            ALT
              from.left ? CASE
                    row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                      to.father ! row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
```

```
                from.right ? CASE
                        row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
                        to.father ! row; row.num; no.of.elements :: [C[row.num] FROM 0 FOR no.of.elements]
    IS.LEAF
            SEQ i = (name * RATIO) FOR RATIO
              to.father ! row; i; MAX.DIM :: [C[i] FROM 0 FOR MAX.DIM]
  :
```

```
  SEQ
    receive(from.father, to.left, to.right)
    multiply()
    send( to.father, from.left, from.right)
:
```

Bibliografía

[Abo90] J. Abolhassan, J. Keller, J. Paul. *On Physical Realizations of the Theoretical PRAM Model.* Technical Report 21/1990. Universität des Saarlandes. SFB 124; 1990.

[Abo91a] F. Abolhassan, J. Keller, W.J. Paul. *On the Cost-Effectiveness and Realization of the Theoretical PRAM Model.* SFB Report 09/1991. Sonderforschungsbereich 124. Fachbereich 14. Universität des Saarlandes. W-6600 Saarbrücken. Germany.

[Abo91b] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, D. Scheerer. *On the physical Design of PRAMs.* Universität des Saarlandes, Computer Science Department. Postfach 1150, 66041 Saarbrücken, Germany, CWI, Dept. AA, Postbus 94079, 1090 GB Amsterdam, The Netherlands.

[Aho74] A. V. Aho, R. Sheti and J. D. Ullman. *Compilers. Principles, techniques and tools.* Addison-Wesley; 1986.

[Aho74] A.V. Aho, J.E. Hopcroft, J.D.Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley; 1974.

[Ajt83] Ajtai, Komlos and Szemeredi. *An O(n log n) Sorting Network.* Combinatoria 3, pp. 1-19; 1983.

[Akl89a] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall - Englewood Cliffs. New Jersey; 1989.

[Akl89b] S.G. Akl y G.R. Guenther. *Broadcasting with Selective Reduction.* Proc. IFIP 11th World Computer Congress, pp. 515-520; 1989.

[Akl92] G.S. Akl, L. Lindon. *Modéles de Calcul Paralléle Mémoire Partagée.* Algorithm paralléle, Algorithmique paralléle. Codonne por M. Cosnard, M. Nivat, Y. Robert. Masson S.A.. Paris 1992. 20th LITP Seing School (Sables-d'Or-les-Pins, 25-29 May 1992).

[Alt87] Alt, Hagerup, Mehlhorn and Preparata. *Deterministic Simulation of Idealized Parallel Computers on more Realistic ones.* SIAM Journal on Computing 16(5), pp. 808-835; 1987.

[Ame78] *American National Standard Programming Language Fortran*, ANSI X3.9-1978. American National Standards Institute; 1978.

[And88] Andenson and Miller. *Optical Communication for Pointer Based Algorithms.* Tech Rep. CR I 88-14. Computer Science Dept., Univ. of Southern California; 1988.

[Ans89] ANSI X3J11 Committee. *ANSI C.* American National Standards Institute (ANSI). 1430 Broadway. New York. NY 10018; 1989.

[Bel91] S.J. Bellantoni. *Parallel Random Access Machines with Bounded Memory Word-Size.* Information and Computation 91 (2), pp. 259-273; 1991.

[Bha] P.C. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, S. Saxena, T. Radzik. *Improved Deterministic Parallel Integer Sorting*. (To appear).

[Ble89] G. E. Blelloch. *Scans as Primitive Parallel Operations.* IEEE Transactions on Computers C-38(11), pp. 1526-1538; 1989.

[Ble94] G.E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, S. Chatterjee. *Implementation of a Portable Nested Data Parallel Languages.* Journal of Parallel and Distributed Computing 21 (1), pp. 4-14; April 1994.

[Bou92] S. Boucheror. *Sur le Modèle d'exècution des PRAM.* Algorithm paralléle, Algorithmique paralléle. Codonne por M. Cosnard, M. Nivat, Y. Robert. Masson S.A., Paris 1992. 20th LITP Seing School (Sables-d'Or-les-Pins, 25-29 May 1992).

[Col89] R. Cole, O. Zajiceck. *The APRAM: Incorporating Asynchrony into the PRAM Model.* Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures, pp. 169-178; 1989.

[Col90] R. Cole, O. Zajiceck. *The Expected Advantage of Asynchrony.* Proc. 2st Annual ACM Symp. Parallel Algorithms and Architectures, pp. 85-94; 1990.

[Con93] B. Condenotti, M. Leoncini. *Introduction to Parallel Processing.* Addison-Wesley; 1993.

[Cul93] D. Culler, R Karp, D. Patterson, A. Sahay, K. Schullser, E. Santos, R. Subramonian, T. vonEicken. *Log P: Towards a Realistic Model of Parallel Computation.* Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP. San Diego. California. Vol.28, pp. 19-22. ACM Press; May 1993.

[Cha92] P. Chaudhuri. *Parallel Algorithms Design and Analysis.* Prentice-Hall; 1992.

[Cha93] F.J. Chávez, B. Said Kayed. *Extensión de los Tipos de Datos del Lenguaje Pascal ll.* Proyecto de Diplomatura. Universidad de La Laguna; 1993.

[Dym87] P.W. Dymond, W.L. Ruzzo. *Parallel PRAMS with Owned Global Memory and Deterministic Context-free Language Recognition.* (extended abstract). Proc. 13th Int. Collq. Automata, Languages and Programming Springer Lecture Notes in Computer Science 226, pp. 95-104; 1987.

[For78] S. Fortune, J. Wyllie. *Parallelism in Random Access Machines.* STOC1978, pp.114-118; 1978.

[Gar94] F. García, C. Rodríguez, C. León, F. Sande. *A High-Level Parallel Language and its Implementation on Transputer Networks.* Proceedings of the Seventh Conference of the North American Transputer User Group. IOSPress, pp. 83-294; 1994.

[Geh88] N.H. Gehani, W. D. Roomme. *Concurrent C..* In N.H. Gehani and A. D. MacGettric, editors. Concurrent Programming, pp.112-141, Addison-Wesley; 1988.

[Gib88] A. Gibbons, W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press; 1988.

[Gib89a] P. B. Gibbons. *A more Practical PRAM Model.* Proc. 1st Annual ACM Symp. Parallel Architectures and Algorithms, pp. 158-168; 1989.

[Gib89b] P.B. Gibbons. *The Asynchronous PRAM: a Semi-synchronous Model for Shared Memory MIMD Machines.* Ph. D. Thesis, Computer Science Division. Univ. California. Berkeley, California 1989.

[Hag91] T. Hagerup, A. Schmitt, H. Seidl. *FORK: A High-Level Language for PRAMs.* PARLE'91; 1990.

[Hag92] T. Hagerup, A. Schmidt, H. Seidl. *FORK: A High-Level Language for PRAMs.* Future Generation Computer Systems 8, pp 379-393; 1992.

[Ham92] P. Hämäläinen. *A PRAM Emulator.* University of Joensun, Finland. Department of Computer Science. Internal Report B-1992-2; 1992.

[Han75] P. B. Hansen. *The Programming Languages Concurrent Pascal.* IEEE Transaction on Software Engenieering 1(2). pp. 199-207, Junio; 1975.

[Han85] P. B. Hansen. *Brinch Hansen on Pascal Compilers.* Prentice Hall; 1985.

[Har96] J.C. Hardwick. *An Efficient Implementation of Nested Data Paralelism for Irregular Divide and Conquer Algorithms.* First International Workshop on High-Level Programming Models and Supportive Environments. IEEE Computer Soc., pp. 105-114; April 1996.

[Hat91] P.J. Hatcher, M.J. Quinn. *Data Parallel Programming on MIMD Computers.* MIT-Press; 1991.

[Her94] C. Hernández. *Un Ensamblador y un Linker para el Sistema ll.* Proyecto de Diplomatura. Universidad de La Laguna; 1994.

[Hey91] T Heywood, S. Ranka. *A Practical Hierarchical Model of Parallel Computation.* Proc. 3rd IEEE Symp. Parallel and Distributed Processing, pp. 18-25; 1991.

[Hid93] M.M. Hidalgo. *Implementación de algunos Algoritmos Aleatorios en ll.* Proyecto de Diplomatura. Universidad de La Laguna; 1993.

[Inm84] Inmos Ltd. *Occam Programing Manual*. Prentice Hall. New Jersey; 1984.

[Inm91] Inmos Ltd. *Occam 2 Toolset User Manual*. INMOS Limited; 1991.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley; 1992.

[Juv92a] S. Juvaste. *An Implementation of the Programing Language pm2 for PRAM.* University of Joensun, Finland. Department of Computer Science. Internal Report A-1992-1; 1992.

[Juv92b] S. Juvaste. *The Programming Language pm2 for PRAM.* University of Joensun, Finland. Department of Computer Science. Internal Report B-1992-1; 1992.

[Kel94] J. Keller, W. J. Paul, D. Scheereer. *Realization of PRAMs: Processor Desing,* In Proc. WDAG94, 8th Int. Workshop on Distributed Algorithms. Springer Lecture Notes in Computer Science. Vol 857, pp. 17-27, 1994. (http://www-wjp.es.uni-sb.de/sbpram)

[Kes94] C. Keßler, H. Seidl. *Making FORK Practical.* Technical Report 1/95, Universität des Saarlandes and Universität Trier SFB 124-C1; 1994.(http://www-wjp.es.uni-sb.de/fork95/fork95.html).

[Kes95a] C. Keßler, H. Seidl. *Fork95 Language and Compiler for the SB-PRAM.* Proceeding of the 5th Int. Workshop on Compiler for Parallel Computers. Málaga; 1995.

[Kes95b] C. Keßler, H. Seidl. *Integrating Synchronous and Asynchronous Paradigms: the Fork95 Parallel Programing Language.* Proceding of MPPM-95. Conference on Massively Parallel Programming Models. Berlin. IEEE CS Press; 1995.

[Kru90] C.P. Kruskal, L. Rudolph, M. Snir. *A Complexity Theory of Efficient Parallel Algorithms.* Theoretical Computer Science 71, pp. 95-132; 1990.

[Lei91] F.T. Leighton. *Introduction to Paralell Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., San Mateo, C.A.; 1992.

[Leo] C. León, C. Rodríguez, F. García, F. de Sande. *A PRAM Oriented Programing System.* CONCURRENCY: Practice & Experience. (Pendiente de Publicación).

[Leo91a] C. León, F. García, C. Rodríguez. *Un Simulador del Modelo PRAM.* XIX Reunión Nacional de Estadística, Investigación Operativa e Informática. Marzo 1991.

[Leo91b] C. León, F. García, C. Rodríguez. *A Parallel Pascal Compiler.* XIX Congreso Nacional de Estadística, Investigación Operativa e Informática. Marzo 1991.

[Leo92] C. León. *Un Compilador Pascal Paralelo para el Modelo PRAM.* Memoria de Licenciatura. Universidad de La Laguna. Enero 1992.

[Leo93] C. León, F. de Sande, F. García, C. Rodríguez y C. Martín. *Un Entorno de Programación para la Enseñanza de Algoritmos Paralelos.* Jornadas sobre NuevasTecnologías en la Enseñanza de las Matemáticas. Valencia. Abril 1.993.

[Leo95a] C. León, F. Sande, C. Rodríguez, F. García. *A PRAM Oriented Language.* Proceedings of the Euromicro Workshop on Parallel and Distributed Processing. IEEE Computer Society Press, pp. 182-191; 1995.

[Leo95b] C. León, F. Sande, F. García, C. Rodríguez. *Implementación de un Entorno Paralelo Orientado al Modelo PRAM.* Actas de las I Jornadas de Informática, pg. 41-51. Puerto de la Cruz. Tenerife; 1995.

[Leo96] C. León, C. Rodríguez, F. Sande, F. García. *Un Estudio Comparativo de Lenguajes Orientados al Modelo PRAM.* Actas de las II Jornadas de Informática, pg. 61-71. Almuñecar. Granada; 1996.

[Li85] K. C. Li and H.Schwetman. *Vector C: A Vector Processing Language.* Journal of Parallel and Distributed Computing, 2, pp. 132-169,1985.

[Lov93] *Draft High Performance Fortran Language Specification, Version 1.0 Draft.* David Loveman, ed.. Tech. Report 92225, CRPC, Rice University. Houston. Texas; January 1993.

[Luc90] F. Luccio, A. Pietracaprina, G. Pucci. *A new Scheme for the Deterministics Simulations of PRAMs in VLSI.* Algorithmica, 5, pp. 529-544; 1990.

[Mar93a] C. Martín. *Técnicas Algorítmicas Paralelas para el Modelo PRAM.* Memoria de Licenciatura. Universidad de La Laguna; Enero 1993.

[Mar93b] D.F. Marqués, R. Acosta. *Algoritmos Paralelos para el Modelo PRAM.* Proyecto de Diplomatura. Universidad de La Laguna; 1993.

[Meh84] K. Mehlhorn, U. Vishkin. *Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories.* Act. Inf. 21, pp. 339-374; 1984.

[Met90] M. Metcalf, J. Reid. *Fortan 90 Explained.* Oxford Science Publications. Oxford. England; 1990.

[Nas93a] J.H. Nash. *A study of the XPRAM Model for Parallel Computing.* PhD. Thesis. University of Leeds; 1993.

[Nas93b] J.H. Nash, P. Dew. *Parallel Algorithms Design on the XPRAM model.* Proceedings of the 22nd Abstract machines Workshop. Leeds; 1993.

[Nas95] J.M. Nash, M.E. Dyer, P.M. Dew. *Designing Practical Parallel Algorithms for Scalable Message Passing Machines.* Transputer Applications and Systems '95, pp. 529-541. IOS Press; 1995.

[Nie93] I. Nieves. *Técnicas de Programación Paralela sobre Listas y Árboles.* Proyecto de Diplomatura. Universidad de La Laguna; 1993.

[Nik85] N. Wirth. *Programming in Modula-2.* Springer Verlag, 3th corrected edition; 1985

[Nis90] N. Nishimura. *Asynchronous Shared Memory Parallel Computation.* Proc. 2nd Annual ACM Symp. Parallel Algorithms and Architecture, pp. 76-84; 1990.

[Par87] I. Parberry. *Some Practical Simulations of Impractical Parallel Computers.* Parallel Computing, 4, pp. 93-101. North-Holland; 1987.

[Qui94] M.J. Quinn. *Parallel Computing. Theory and Practice*. McGraw-Hill, INC. 2nd Edition; 1994.

[Ran89a] A. Ranade. *Fluent Parallel Computation.* PhD. Yale University, Dep. of Computer Science, May 1989.

[Ran89b] A.G. Ranade. *How to Emulate Shared Memory.* Journal of Computer and Systems Sciences, 42(3), pp 307-326; 1991.

[Rog92] G. Röger, K. Sieber. *A Traced-Based Denotational Semantics for the PRAM-Language FORK.* 1/1992, SFB 124-C1. FB 14, Universität des Saarlandes, 6600 Saarbrüken, Germany.

[Ros87] J. Rose, G. Steele. *C*: An Extended C Language for Data Parallel Programing.* Technical Report PL 87-5, Thinking Machines Corporation; 1987.

[Ros90] P. Rossmanith. *The Owner Concept for PRAMs.* SFB 343/15/90. Institut fur Informatik. Technische Universitat Munchen. Munchen. Germany; 1990.

[San] F. de Sande, F. García, C. León, C. Rodríguez. *The ll Programing Language.* IEEE Transactions on Education. (Pendiente de Publicación).

[San93] F. Sande. *Realización de un Entorno de Programación Paralela de Alto Nivel sobre Redes de Transputers.* Memoria de Licenciatura. Universidad de La Laguna; 1993.

[Sch91] A. Schmitt. *A Formal Semantics of FORK.* Internal Report, May 1991. Fachbereich 14, Universitä des Saarlandes, Im Stadtwald, 6600 Saarbrüken, Germany.

[Sch92] J. Schlesinger, M. Gokhale. *DBC Reference Manual.* Technical Report TR-92-068 Supercomputing Research Center. 1992.

[Sto84] L. Stockmeyer, U. Vishkin. *Simulation of Parallel Random Access Machines by circuits.* SIAM, Journal on Computing, Vol.13, No.2 May. 1984.

[Tic90] W.F. Tichy, C.G. Herter. *Modula-2: An Extension of Modula-2 for Highly Parallel Portable Programs.* Interner Bericht Nr. 4/90; January 1990.

[Uni83] United States Department of Defense. *Reference Manual for ADA programming language,* ANSI/MIL-STD-1815A.; 1983.

[Val90a] L.G. Valiant. *A Bridging Model for Parallel Computation.* Communications of the ACM, Vol.33, No.8, pp.103-111; August 1990.

[Val90b] L.G.Valiant. *General Purpose Parallel Architectures.* Handbook of Theoretical Computer Science. J. van Leeuwen de. MIT Press. Chapter 18, pp. 943-971; 1990.

[Vis84] U. Vishkin. *A Parallel Design Distributed Implementation (PDDI) General Purpose Computer.* Theoretical Computer Science 32, pp. 157-172; 1984.

[Vis85] U. Vishkin, A. Wigderson. *Trade-offs between Depth and width in Parallel Computation.* SIAM Journal on Computing 14(2), pp. 303-314; 1985.

[Wel94] P.H. Welch. *Parallel Hardware and Parallel Software: a Reconciliation*. Computing Laboratory. University of Kent at Canterbury, CT2 7NF, England; 1994.