Procesador Paralelo de Prolog sobre una Arquitectura de Memoria Distribuida

Memoria que presenta para optar al grado de
Doctor en Ciencias Físicas

Lourdes Araujo Serna

Dirigida por el profesor

Jose Jaime Ruz Ortiz

Departamento de Informática y Automática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid

Mayo 1994

a mis padres

Agradecimientos

Hay mucha gente a la que me gustaría agradecer su apoyo durante la realización de esta tesis. Quiero empezar a hacerlo por mis padres que han hecho posible que terminase el trabajo y me han animado a ello.

Quiero expresar un agradecimiento muy especial a mi director de tesis Jose Ruz, no sólo por la iniciación en el tema, sino también por animarme a entrar en la universidad, así como por su dedicación y apoyo en el desarrollo del trabajo, estando siempre dispuesto a investigar nuevos caminos.

Quiero agradecer a Jose Cuesta su apoyo y ayuda durante todos los años en los que he estado realizando este trabajo, así como su paciencia, ya que quizás ha tenido que escuchar sobre Prolog y paralelismo más de lo que probablemente hubiese querido.

Estoy especialmente agradecida a Mario Artalejo, Ana Gil y Teresa Hortalá por haberme animado y prestado su ayuda constantemente. Fernando Saenz y Alvaro Urech han estado siempre dispuestos a escuchar y ayudar. Mis compañeros de fatigas en los últimos meses Juan Carlos Gonzalez y Paco Lopez han hecho la tarea más llevadera. También quiero agradecer a Puri Arenas, que ha sido mi compañera de despacho durante el último año los buenos ratos que hemos pasado a pesar de los agobios de última hora. Otras personas del departamento cuya ayuda ha sido decisiva han sido Victor Sanchez y Alfredo Bautista, que me introdujeron en el uso del Supernodo y sus misterios. También quiero recordar a mis amigos de Inglaterra con los que empecé a estudiar los transputers, Innes Jelly y Ross MacMillan por su atención y hospitalidad.

También quiero dar las gracias a la gente que *iba* al Tomi, Angel Pascual, Angel Sanchez, Jose Cuesta, Jose Olarrea, Leonor Tabernero, Rodrigo Moreno, Susana Gilardi, Rubén Perez, Rosalia Serna, Ricardo Brito, Pepe Aranda y Carmen Pereña, a los que espero acompañar más a menudo a

partir de ahora.

Lurdes Araujo Serna
Madrid, 16 de mayo de 1994

Indice

Indice de Tablas

Indice de Figuras

Tweedledee:

Si así fuese, así podría ser;
si así fuese, así sería;
pero como no es, no es;
¡eso es lógica!

Lewis Carrol. *A través del espejo.*

Prólogo

Prolog es un lenguaje de programación basado en la lógica de primer orden especialmente adecuado para expresar aplicaciones de tipo simbólico, como bases de datos, sistemas expertos y demostración automática. La eficiencia de las implementaciones de Prolog ha venido mejorando desde la aparición de la Máquina abstracta de Warren (WAM), que ha acortado la distancia entre la semántica declarativa del lenguaje y las arquitecturas convencionales de tipo Von Newman sobre las que se implementa. No obstante, las optimizaciones de la WAM tienen un límite determinado por su naturaleza secuencial, que solo se puede superar con modelos de ejecución paralela. La combinación del poder descriptivo de la lógica con el alto rendimiento potencial de los sistemas paralelos tiene ventajas en ambas areas: la lógica permite la expresión del paralelismo de una forma natural y los sistemas paralelos proporcionan una rápida plataforma de ejecución de los programas lógicos. El paralelismo de un programa lógico se manifiesta principalmente de dos formas: *Paralelismo_Y* y *paralelismo_O*. El primero ejecuta simultaneamente los objetivos de la pregunta o del cuerpo de las cláusulas, siendo la principal dificultad para su explotación los conflictos de vinculación de las variables lógicas compartidas. Una alternativa para solucionar este problema es considerar únicamente la ejecución paralela de los objetivos cuando son independientes (*Paralelismo_Y Independientes*). El paralelismo_O es el resultado del indeterminismo de los programs lógicos y realiza la computación simultanea de un objetivo con las distintas cláusulas del procedimiento asociado.

Propósito del trabajo

El propósito de este trabajo ha sido el desarrollo de un modelo de ejecución paralelo de los programas lógicos para sistemas distribuidos, su implementación y su evaluación. El modelo considera la explotación del par-

alelismo_Y independiente y el paralelismo_O, así como la combinación de ambos. El diseño está basado en las siguientes ideas:

- El sistema está soportado por una **arquitectura distribuida** con un alto número de procesadores que trabajan bajo un **control jerárquico**. El control se realiza en una serie de procesadores llamados *controladores*. El resto de los procesadores, *procesadores básicos*, están dedicados a la ejecución de programas Prolog.

- **Sistema multisecuencial** en el que las computaciones que se realizan en paralelo siguen el esquema básico del modelo de ejecución secuencial.

- El **paralelismo_Y** se explota siguiendo un modelo de **entornos cerrados** (sin referencias a variables externas) en el que se forman *tareas_Y* autónomas para la computación de cada objetivo independiente.

- La explotación del **paralelismo_O** se basa en la ejecución multisecuencial de las ramas del árbol de búsqueda. Cada *tarea_O* nueva que se crea para la exploración de una nueva rama reconstruye el entorno de la tarea padre **recomputando** el objetivo inicial pero siguiendo el *camino de éxito* (sin backtracking) recibido de la tarea padre.

- Cuando un programa presenta **paralelismo O_bajo_Y** y es necesario combinar las distintas soluciones de los objetivos paralelos, se evita el *almacenamiento de soluciones parciales y la sincronización de tareas* produciendo la combinación **de forma distribuída**. No hay una tarea encargada de realizar el producto cruzado de las distintas alternativas, sino que cada nueva tarea que se crea conoce automáticamente que combinaciones de soluciones le corresponden. La idea es crear una computación para cada combinación de soluciones, recomputando el camino de éxito que lleva del objetivo inicial a la llamada paralela considerada. De esta forma la explotación del paralelismo_Y se realiza con el mecanismo del paralelismo_O, creando tareas autónomas que reducen considerablemente el tráfico de mensajes.

Se ha diseñado una máquina abstracta paralela que implementa el modelo de ejecución de los procesadores básicos del sistema. Esta máquina es una extensión de la WAM y mantiene sus técnicas de ejecución en los segmentos secuenciales de programa conservando así las optimizaciones ya

conseguidas en Prolog secuencial. De la misma forma se ha tratado de mantener las técnicas de explotación de cada tipo de paralelismo con las menores modificaciones.

El estudio de la planificación ha sido otro objetivo fundamental en el trabajo. Se han evaluando distintas políticas de reparto del trabajo pendiente entre los procesadores desocupados. También se ha considerado la granularidad de los trabajos pendientes, diseñando distintos controles de granularidad.

Finalmente se ha realizado una implementación sobre un sistema de transputers, haciendo un estudio de la red de interconexión que soporta al sistema. Se ha evaluado la explotación de cada tipo de paralelismo y su combinación.

Organización de los siguientes capítulos

El trabajo se organiza como sigue:

El Capítulo 1 hace un breve recorrido por los fundamentos de la programación lógica, presentando las características más relevantes de Prolog. A continuación describe el modelo de implementación secuencial de la WAM. Y finalmente describe los tipos de paralelismo que presentan los programas Prolog, dando una panorámica de los trabajos realizados en este area.

El Capítulo 2 describe el modelo de ejecución del *Procesador Distribuido de Prolog (PDP)* diseñado. Se describe el modelo de explotación del Paralelismo_Y por el método de entornos cerrados y la extensión del mecanismo de backtracking que requiere el modelo. A continuación se presenta el modelo de explotación del Paralelismo_O utilizando mecanismos de copia y recomputación. Finalmente se describe el modelo de ejecución de PDP que introduce un mecanismo de combinación de soluciones para la explotación del Paralelismo O_bajo_Y basado en la recomputación y en la creación de computaciones autónomas. Se introduce también una representación de la ejecución paralela de un programa en forma de árbol de tareas.

El Capítulo 3 describe la implementación del modelo de ejecución de los procesadores básicos de PDP. Se describe su arquitectura, es decir, las estructuras de datos e instrucciones que han aparecido como consecuencia de la explotación del paralelismo. Se describen también los procedimientos de intercambio de trabajo entre procesadores básicos.

El Capítulo 4 describe distintos aspectos de la planificación del trabajo: el establecimiento de las unidades de división de los trabajo, la selección de los trabajos a compartir por los procesadores básicos y la asignación por los controladores de los trabajos seleccionados a los procesadores.

El Capítulo 5 describe detalles de la implementación del sistema sobre una red de transputers. Se ha estudiado la topología de la red más apropiada para PDP, optando por una red toroidal. Se presenta también el procedimiento de encaminamiento de mensajes necesario para la topología elegida. Finalmente se presentan los detalles del desarrollo de PDP, incluyendo el protocolo de comunicaciones.

En el Capítulo 6 se evaluan las ideas presentandas en los anteriores. Para cada tipo de paralelismo se han investigado los siguientes paramétros coma la **curva de mejora del rendimiento** y su relación con el paralelismo medio de los programas, el recargo que introduce su explotación y el reparto del tiempo entre la actividades que desarrolla un procesador al explotar el tipo de paralelismo. Además, se presentan otros resultados particulares de cada tipo de paralelismo. Finalmente, se comparan los resultados obtenidos con los de otros sistemas.

Finalmente, el Capítulo 7 presenta las conclusiones y sugerencias para futuros trabajos.

Los apendices A, B y C describen el simulador construido para la evaluación inicial de la explotación del Paralelismo_Y, una propuesta de implementación de los predicados extralógicos de Prolog y la validación del protocolo de comunicaciones del sistema.

1

Introducción

Este capítulo hace un breve recorrido por los fundamentos de la programación lógica, presentando las características más relevantes de su representante más difundido: el lenguaje Prolog. A continuación se describe el modelo de implementación secuencial más extendido, la Máquina Abstracta de Warren, que ha permitido salvar la distancia entre la semántica declarativa de los lenguajes lógicos y las arquitecturas de computadores convencionales. Finalmente, se describen los tipos de paralelismo que presentan los programas Prolog, dando una panorámica de los trabajos realizados en este area.

1.1 Fundamentos de la Programación Lógica

La programación lógica surgió a partir de investigaciones en demostración automática, en particular de los trabajos sobre resolución de Robinson [55]. El principio de resolución es una regla de inferencia, junto con una potente operación de encaje de patrones que es la *unificación*. Green [31] mostró como usar un sistema de resolución implementado en LISP, para simular la computación de un programa. Los trabajos de Boyer y Moore [14] mostraron como las cláusulas derivadas en una resolución podian representarse por punteros a las cláusulas iniciles y entorno de vinculación, es decir, una representación similar a la de los registros de activación de las implementaciones de los lenguajes imperativos. Los trabajos de Kowalski [41] llevaron al diseño de un sistema de inferencia basado en la resolución, la resolución SLD, más apropiada para la programación, que lleva a cabo la deducción de una forma dirigida por el objetivo. Finalmente, Colmerauer

y Roussel [24] diseñaron e implementaron el lenguaje Prolog basado en las ideas anteriores. Existen diversos trabajos que presentan un tratamiento formal de la semántica de la programación lógica [8] [49].

Un *programa lógico* es un conjunto de cláusulas de la forma:

$$A \longleftarrow B_1, ..., B_n$$

donde $A, B_1, ..., B_n$ son átomos (relaciones n-arias cuyos argumentos son términos: variables, constantes o funciones), A es la *conclusión* y B_1,...,B_k son las premisas. Cuando el conjunto de conclusiones es vacio la cláusula se denomina *objetivo*. Los programas lógicos se evaluan por medio de la *resolución SLD* (Resolución Lineal con Regla de Selección para Cláusulas definidas). Una *sustitución*, $\theta = \{x_1/t_1, ..., x_n/t_n\}$, es una ligadura de variables a términos. Si toda t_i carece de varibles entonces θ es *cerrada*. *Restricción* de una sustitución θ a un conjunto de varibles X es la sustitución formada por las ligaduras x_i/t_i de θ para las que $x_i \in X$. Las sustituciones operan sobre expresiones, siendo $E\theta$ el resultado que se obtiene al reemplazar simultaneamente cada ocurrencia en E de las variables de θ por el término correspondiente. Dados dos átomos A y B, si para una sustitución θ se cumple $A\theta \equiv B\theta$ entonces θ es un *unificador* de A y B y se dice que A y B son *unificables*. En base al teorema de unificación debido a Robinson sabemos que existe un algoritmo (el algoritmo de unificación) que para dos átomos cualesquiera produce su unificador más general (umg) si son unificables y en caso contrario informa de la no exitencia del unificador.

Dado un programa lógico P y un objetivo $N = \leftarrow A_1, ..., A_n$, si $C = A \leftarrow B_1, ..., B_k$ es una cláusula de P entonces si para algún i, $1 \leq i \leq n$, A_i y A unifican con un umg θ,

$$N' = \leftarrow (A_i, ..., A_{i-1}, B_1, ..., B_k, A_{i+1}, ..., A_n)\theta$$

es un *resolvente* de N y C. Iterando el proceso de obtención de resolventes se consigue una secuencia de resolventes llamada *derivación*. Derivación_SLD de un programa P y un resolvente N es una secuencia $N = N_0, N_1, ...$ de objetivos junto con una secuencia $C_0, C_1, ...$ de variantes de cláusulas de P y una secuencia $\theta_0, \theta_1, ...$ tales que para todo $i = 0, 1, ...$

a) N_{i+1} es un resolvente de N_i y C_i,

b) θ_i es el umg del paso d) de la secuencia para obtener un resolvente

c) C_i no tiene variables en común con N_i

La restricción de $\theta_0, ..., \theta_m$ a las varibles de N es la *sustitución de respuesta computada* para $P \cup \{N\}$. Cuando uno de los resolventes N_i es vacio $\square$ se ha llegado a la última cláusula negativa de la derivación y se ha demostrado que el conjunto formado por el programa y la negación del objetivo que queremos satisfacer es inconsistente, y por consiguiente el objetivo se deriva del programa. La totalidad de las derivaciones forman el *espacio de búsqueda*, que suele representarse en forma de árbol. Las rámas que parten de un nodo representan las alternativas posibles de átomos del resolvente (nodo_Y) o de cláusulas del programa que unifican con el átomo seleccionado (nodo_O).

1.2 El lenguaje Prolog

Para implementar un lenguaje de programación basado en el modelo computacional de la programación lógica es necesario automatizar el mecanismo descrito, fijando unas reglas para los pasos en que hay que realizar alguna selección. Prolog utiliza las siguientes reglas de selección:

- Al seleccionar un átomo del resolvente a computar, se toma el que se encuentra más a la izquierda según el orden de aparición en el programa
- Al seleccionar una cláusula para unificar con el átomo que se está computando, se toma la primera cuya cabeza unifica con el átomo seleccionado en el orden textual del programa. Si esta elección no permite obtener una solución, se selecciona la siguiente cláusula cuya cabeza unifica con el objetivo (se hace *backtracking*).

A continuación introducimos una serie de términos para designar los objetos que nos van a permitir esquematizar la semántica operacional de la ejecución de un programa. Un *programa* $\wp$ representa a un programa Prolog traducido a código intermedio por el compilador. Un *procedimiento* asociado a un átomo i, P_i, es el conjunto de cláusulas de un predicado que quedan por seleccionar para unificar con un átomo. En la *pila de backtracking* S se almacenan los *resolventes* R, *procedimientos asociados* P_i y *sustituciones* V que se producen a lo largo de la ejecución. La ejecución se describe por el esquema de la Figura 1.2.

La ejecución se inicia teniendo como resolvente el objetivo Q a resolver y termina, con éxito, cuando el resolvente queda vacio ó, con fallo, cuando sin

```
type tipo_res = (exito, fallo);
proccess crear_ejecucion(Q:objetivo;p:programa);
var
    R : resolvente;
    S : pila de backtracking;
    V : sustitución;
begin
    R := [Q];
    S := ∅;
    V := ∅;
    ejecucion(p, R, S, V);
end

procedure ejecucion(p:programa; R:resolvente; S:pila de backtracking: V:sustitucion);
var
    P_a: procedimiento asociado al predicado a;
    resultado:tipo_res;
    a : objetivo;
    c : clausula;
    V' : sustitución;
begin
   repeat
       a := R[1] /* Se toma el primer atomo del resolvente */
       buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
       repeat
           c := P_a[1]; /* Se toma la primera clausula del procedimiento */
           P_a := P_a - c;
           resultado := unificar(a,c,V');
       until (P_a = []) or resultado = exito;
       if resultado = exito then
       begin
           if P_a <> [ ] then salvar(P_a,R,V,S);
           V := componer(V,V');
           reemplazar(R,a,c); /* En R, reemplazar a por el cuerpo de c */
       end
       else
           resultado := backtracking(S,a,P_a,R,V);
    until (R = []) or resultado = fallo;
    if resultado = exito then begin
       V := restricción(V,Q);
       visualizar(V);
    end
end
```

Figura 1.1: Esquema de la ejecución de Prolog

estar vacio el resolvente ya se han probado sin éxito todas cláusulas alternativas. En cada paso de la resolución se toma el primer átomo del resolvente a y se busca el procedimiento asociado P_a (*buscar_procedimiento*). Se intenta entonces la unificación con las sucesivas cláusulas del procedimiento hasta obtener éxito. La unificación produce un conjunto de nuevas vinculaciones V'. Si han quedado clausulas del procedimiento sin probar se almacenan junto con el resolvente y la sustitución producida en la pila de backtracking S (*salvar*) y se actualizan la sustitución V con las nuevas vinculaciones V' (*componer*) y el resolvente R. Si no se consigue la unificación se produce backtracking, recuperando de la pila estados anteriores. Si esto no es posible por estar la pila S vacia la ejecución termina con fallo. Cuando la ejecución termina con éxito, se devuelve la sustitución de respuesta computada, que se construye como la restricción de las sustituciones producidas durante la resolución sobre las variables del objetivo.

Podemos describir la aplicación de backtracking con las siguientes acciones:

- Se eliminan las sustituciones producidas a partir de la elección que se va a cambiar.

- Se selecciona la cláusula siguiente del procedimiento asociado al átomo que se va a volver a unificar. Esta cláusula se elimina del procedimiento.

El esquema del mecanismo de backtracking aparece en la figure 3.18. Se *saca* de la pila de backtracking el último resolvente almacenado, cuyo primer átomo es el objetivo a reevaluar. Se *saca* también de la pila de backtracking las cláusulas que quedan sin probar, P_a, del procedimiento asociado al objetivo a, así como las vinculaciones V, que se tenian cuando se ejecutó este objetivo por primera vez. De esta forma, se deshacen las vinculaciones producidas después del objetivo. Una vez restaurado el estado de ejecución de esta forma, se *reevalua* el objetivo. La función de reevaluación es una ejecución de un objetivo, que en lugar de buscar el procedimiento asociado para resolverlo, considera solo aquellas cláusulas de procedimiento que están sin probar y que recibe como parametro.

Otra cuestión a resolver para convertir la programación lógica en un lenguaje práctico es la incorporación de las facilidades del computador como la operaciones de entrada/salida y las funciones aritméticas. Prolog incorpora predicados metalógicos para estas tareas, además de otros que incluye

```
procedure backtracking(p: programa; S:pila de backtracking;
        R:Resolvente; V:sustitución; resultado:tipo_res);
begin
    if not vacior(S) then
        sacar(R,S);
        /* Se toma el primer atomo del resolvente R = [R1, ..., Rm] */
        a := R[1]
        sacar(Pa,S);
        sacar(V,S);
        reevaluacion(p, a, Pa, R, S, V, resultado);
    else resultado := fallo
end
procedure reevaluacion(p: programa; a:objetivo; Pa: procedimiento asociado a a;
                S:pila de backtracking; R:resolvente; V:sustitución; resultado:tipo_res);
begin
   repeat
        c := Pa[1]; /* Se toma la primera clausula del procedimiento */
        Pa := Pa - c;
        resultado := unificar(a,c,V');
    until (Pa = []) or resultado = exito;
    if resultado = exito then begin
        if Pa <> [ ] then salvar(Pa,R,V,S);
        V := componer(V,V');
        ejecucion(p,R,S,V,resultado);
    end
end
```

Figura 1.2: Mecanismo de backtracking

para mejorar la eficiencia o facilitar la programación. Así se tiene el predicado *cut* que permite controlar el *backtracking* eliminando las alternativas pendientes posteriores a la cláusula en la que se encuentra el corte, o los predicados *assert* y *retract* permiten añadir o quitar cláusulas del programa dinámicamente. Aunque estos predicados son muy utiles, rompen la semántica declarativa del lenguaje. Prolog también incorpora facilidades para el manejo de listas, aunque siempre pueden expresarse como términos estructurados.

1.2.1 Sintaxis de Prolog

La eficiente estrategia de resolución de Prolog es posible gracias al limitado número de tipos de cláusulas que acepta:

- Una cláusula con una sola conclusión
  Prepresenta una regla y se expresa:
  p :- $q_1, \ldots, q_n$

- Una cláusula sin premisas
  Representa un hecho y se expresa en la forma:
  p.

- Un objetivo o cláusula negativa
  Representa una pregunta y se expresa:
  :- p.

La ejecución de Prolog funciona como un demostrador de teoremas sencillo. Dada una pregunta y un conjunto de reglas, Prolog intenta construir valores para las variables de la pregunta que la hacen cierta.

1.3 Implementación secuencial de Prolog: la WAM

Las primeras implementaciones de Prolog fueron interpretes que aunque mostraban la potencia del lenguaje no eran suficientemente eficientes para permitir su aplicación a problemas prácticos. En 1983 Warren [70] propuso en su tesis un modelo de ejecución compuesto por una máquina abstracta y una técnica de compilación que se conocen como WAM (Warren Abstract Machine). La WAM a acortado la distancia entre la semántica declarativa de Prolog y la arquitectura convencional Von Newman de los computadores

actuales, identificando los recursos expresivos de Prolog con las construcciones imperativas que soporta la arquitectura y tratando la recursión de los programas con el gasto de las construcciones imperativas.

La WAM es una máquina abstracta consistente en la definición de una arquitectura y un conjunto de instrucciones hechas a la medida de Prolog. Puede implementarse eficientemente en un amplio rango de máquinas reales y actualmente está aceptada como un estandard para la implementación de Prolog.

La WAM, que se describe detalladamente en [1], presenta las siguientes características:

- Implementa la unificación
  La unificación de un átomo con la cabeza de una cláusula se implementa como la llamada a un procedimiento, reservando espacio (*entorno*) en memoria para la ejecución del procedimiento (instrucción *alloc*), preparando los parametros de llamada (instrucciones *put* y *bld* y realizando la llamada (instrucción *call*). Si la unificación (instrucciones *get* y *unify*) tiene éxito se pasa realizar la resolución, resolviendo los objetivos del cuerpo de la cláusula. Al finalizar la resolución se devuelve el control al punto en que se realizó la llamada (intrucción *proceed*) y se libera el espacio reservado (instrucción *dealloc*).

- Considera las alternativas de un procedimiento automáticamente
  Cada procedimiento está formado por el código correspondiente a cada una de sus cláusulas. Cuando el control del programa llega a un procedimiento, se crea un *punto de elección* para realizar *backtracking* automáticamente en caso de que falle la unificación (instrucciónes *try*.)

- Manipula cuatro tipos de datos
  que se corresponden con los tipos de términos Prolog: constantes, variables, estructuras y listas, existiendo instrucciones especializadas en el tratamiento de cada uno de ellos.

- Realiza una unificación especializada
  Con información obtenida en tiempo de compilación, muchos casos de unificación pueden ser tratados como casos particulares, dependiendo del tipo de dato a unificar (constantes, estructuras o listas), de manera que se realiza una unificación más eficiente. Las variables libres se unifican entre ellas y a partir de ese momento se instancian a el mismo valor, siendo necesaria una operación de *desreferenciación* para alcanzarlo.

1.3.1 Areas de datos

La memoria de la WAM está formada por tres areas de datos principales, que se manipulan como *pilas*:

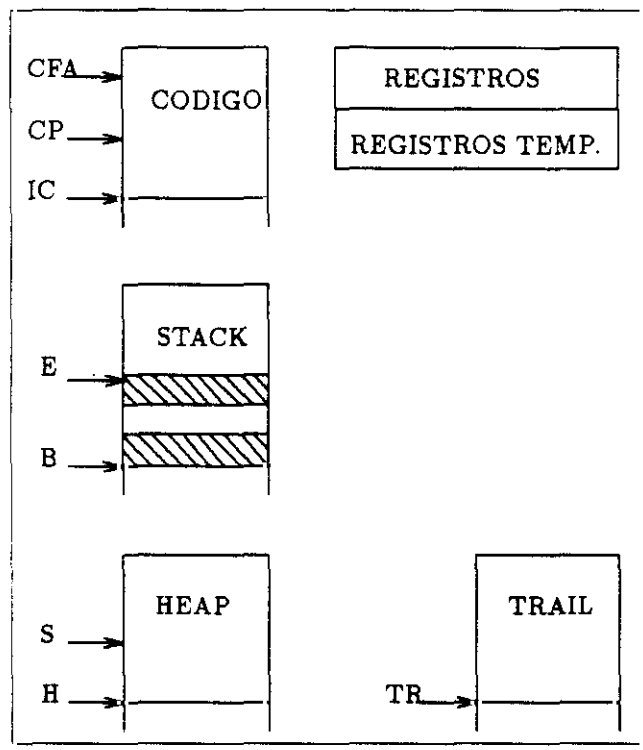


Figura 1.3: Estructuras de datos de la WAM

- *Stack*
  Contine dos tipos de objetos: *entornos* y *puntos de elección*. Los entornos se asocian con la ejecución de una cláuslula con más de un objetivo. Los puntos de elección se crean antes de intentar unificar con un procedimiento que tiene múltiples cláusulas.

- *Heap*
  Contine los términos compuestos: estructuras y listas.

- *Trail*
  Se usa para almacenar la posición de las variables que deben ser desvin-

culadas en caso de *backtracking*.

Además de estas areas de datos dinámicas, la WAM posee una area estática para almacenar el programa y una serie de registros para el paso de parametros:

$$A_1, ..., A_N$$

y para el control de las áreas de datos:

- IC: contador de programa
- RetReg: Registro de retorno
- EnvReg: Registro de entorno
- BReg: Registro de puntos de elección
- TR: Cima de la pila *stack*
- HReg: Cima de la pila *Heap*
- SReg: Registro de estructuras

Los registros *IC* y *RetReg* se utilizan para recorrer el código. IC contiene la dirección de la instrucción que se está ejecutando. RetReg contiene la dirección de vuelta de la llamada. *EnvReg* y *BReg* apuntan al último entorno y al último punto de elección del *stack* respectivamente. TR apunta a la cima de la pila *Trail*. *HReg* y *SReg* apuntan al *Heap*. *HReg* apunta a la cima del *Heap*, mientras qeu *SReg* se utiliza para recorrer las estructura durante la unificación.

Los *entornos* (Figura 1.4) almacenan las variables permanentes de la cláusula actual de un procedimiento y también la dirección de retorno para actualizar *RetReg* cuando sea devuelto el control.

Los *puntos de elección* (Figura 1.3.1) almacenan el estado de la WAM a la entrada de los procedimientos. De esta manera si la unificación con la cláusula actual falla se considera la siguiente, restaurando el estado previamente.

1.3.2 Instrucciones de la WAM

El programa Prolog se compila a una representación intermedia formada por las instrucciones que interpreta la WAM. El conjunto de instrucciones

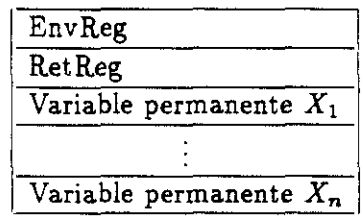


Figura 1.4: Entorno

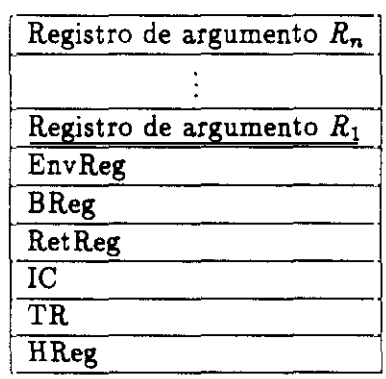


Figura 1.5: Punto de Elección

de la WAM puede clasificarse en instrucciones *put*, *get*, *unify*, instrucciones de control y *switch*.

- Las instruciones *put* se encargan de preparar los parametros de llamada para el próximo objetivo a resolver.
- Las instrucciones *bld* preparan los parametros de llamada que son argumentos de datos estructurados.
- Las instrucciones *unify* realizan la unificación de los argumentos de dtoa estructurados.
- Las instrucciones *get* realizan la unificación de los argumentos del objetivos de llamada con los de la cabeza de la cláusula con la que se intenta la unificación. Estas instrucciones están especilizadas para los tipos de datos constantes, listas y estructuras.
- Las instrucciones de *control* se encargan de preparar y llevar a cabo los saltos de control de programa.
- Las instrucciones *switch* explotan el determinismo del programa ignorando las clausulas sin posibilidad de unificar con el objetivo.

En la tabla 3.1 apacecen las instrucciones junto con una pequeña descripción de su función.

1.3.3 Optimizaciones

La WAM incorpora una serie de optimizaciones al mecanismo básico presentado, algunas de las cuales son las siguientes:

- **Indexación**
  Las instrucciones para manejar el indeterminismo imponen una búsqueda estrictamente secuencial sobre la lista de cláusulas que constituyen una definición. Cuando al menos algunos de los argumentos del predicado de llamada está instanciado, esta información puede utilizarse para acceder a la cabeza de las cláusulas más directamente. La WAM dispone de instrucciones *switch* que permiten ignorar aquellas cláusulas de un predicado que no pueden unificar con un objetivo, debido al tipo de los argumentos que poseen.

Instrucciones put (antes de la llamada)	
putCon ri, c	pone el valor de la constante c en ri
putVar ri, X	crea una nuevavariable en X y la pone en ri
putVal ri, X	mueve X a ri
putStr ri, f	crea un functor f y lo pone en ri
putNil ri	pone NIL en ri
putLis ri	crea una lista y la pone en ri
Instrucciones bld(cargan argumentos de estructuras	
bld_ ri, (...)	carga ri con un argumento de estructura
Instrucciones get (unifican con los registros de argumento)	
get_ ri, (...)	unifica (...) con ri
Instrucciones unify (unifican con argumentos de estructuras	
unify_ (...)	unifica (...) con un argumento de estructura
Instrucciones de control	
call	llama a un predicado
exec	salta a un predicado
alloc	crea un entorno
dealloc	elimina un entorno
proceed	vuelve de un predicado
Instrucciones de manejo del indeterminismo	
tryMeElse	crea un punto de elección
retryMeElse	cambia la direccion de la instrucción a probar
trustMeElseFail	elimina un punto de elcción
Instrucciones de selección de cláusula	
switch_on_term X, C, L, S	salta segun el contenido de r0
switch_on_constant N, tbl	salta segun el contenido de r0
switch_on_struct N, tbl	salta segun el contenido de r0

Tabla 1.1: Conjunto de instrucciones de la WAM

- **Utilización de registros temporales**
  Una variable temporal en una cláusula Prolog es una variable que hace su primera aparición en la cabeza de la clausula, en un término compuesto o en el último objetivo, y que no aparece en más de un objetivo. La cabeza se cuenta como parte del primer objetivo. Estas variables pueden almacenarse en registros en lugar se reservarlas espacio de memoria, con lo que la eficacia de los programas puede mejorar significativamente.

- **Optimización de la última llamada**
  Se trata de una generalización de la optimización de la recursión de cola. Se basa en el hecho de que el espacio asignado a las variables permanentes asociadas a una regla no vuelven a necesitarse después de las instrucciones *put* que preceden a la última llamada del cuerpo. Por tanto puede liberarse el entorno antes de llamar al último procedimiento del cuerpo de la regla.

- **Entornos superfluos**
  Las variables permanentes del entorno actual solo se necesitan hasta que se ejecutan las instrucciones *put* de los argumentos de la última llamada. Esto permite reutilizar el espacio de pila usado para cada llamada en el cuerpo de una regla ya que el entorno puede liberarse antes de realizar la llamada.

1.3.4 Ejemplo de un programa compilado

Ejemplificamos las instrucciones descritas con el programa de la concatenación:

```
concatenar(nil,L,L).
concatenar(cons(X,L1), L2, cons(X,L3)) :- concatenar(L1, L2, L3).

concatenar/3_0
      tryMeElse 3, concatenar/3_1   concatenar
      alloc                                (
      getCon r0, nil                        nil,
      getVar r1, 0                             L,
      getVal r2, 0                               L
      dealloc                                     )      n
      proceed
concatenar/3_1
      trustMeElseFail               concatenar
```

```
alloc                                    (
getStr r0, cons/2                         cons(
uniVar 0                                       X,
uniVar 1                                         L1),
getVar r1, 2                              L2,
getStr r2, cons/2                         cons(
uniVal 0                                       X,
uniVar 3                                         L3)
putVal r0, 1                  L1,
putVal r1, 2                     L2,
putVal r2, 3                        L3
dealloc
exec concatenar/3_0
```

Si ahora planteamos el siguiente objetivo:

```
:- concatenar(cons(1,cons(2,nil)),cons(3,cons(4,nil)),X).
```

```
punto_entrada
      putTStr rt0, cons/2              cons(        -|
      bldCon 2                              2,       |<-|
      bldCon nil                              nil),- |  |
      putStr r0, cons/2           cons(                |
      bldCon 1                         1,              |
      bldTVal rt0                        *-------------|
      putTStr rt0, cons/2               cons(       -|
      bldCon 4                               4,      |<-|
      bldCon nil                               nil),-|  |
      putStr r1, cons/2           cons(                |
      bldCon 3                         3,              |
      bldTVal rt0                        *-------------|
      putVar r2, 0                X
      call concatenar/3_0, 1
      stop
```

1.3.5 Mejorando la eficiencia

A pesar del avance que ha supuesto la WAM en la eficacia de las implementaciones de Prolog, aún no se ha alcanzado el rendimiento de los lenguajes imperativos en la ejecución de muchos programas. Las lineas principales que se investigan para alcanzar esta eficiencia son:

- **Mejorar el modelo secuencial**, limitando las caracteríticas más potentes (y costosas) de la programación lógica a los casos estrictamente necesarios [68]: instrucciones de grano más fino, técnicas de explotación del determinismo en los programas, técnicas para unificación especilizada, etc
- **Explotar los distintos tipos de paralelismo** que los programas lógicos presentan.

Estas direcciones de investigación pueden complementarse para mejorar los resultados. Nosotros nos centramos en la segunda, por lo que en la siguiente sección se hace un recorrido por algunos de los modelos de explotación del paralelismo más difundidos [36] [39].

1.4 Ejecución paralela de Prolog

Existen dos fuentes principales de paralelismo en un programa Prolog: el paralelismo_Y, que consiste en la ejecución simultanea de los objetivos del cuerpo de una cláusula, y el paralelismo_O, que es el resultado del no determinismo de los programas lógicos y se explota probando simultaneamente distintas cláusulas de un procedimiento.

1.4.1 Paralelismo_Y

La explotación del paralelismo_Y se basa en la ejecución simultanea de un conjunto de átomos de un resolvente de la derivación. Si seleccionamos y unificamos simultaneamente un conjunto de átomos se producirá una sustitución θ_i por cada átomo A_i seleccionado. Para mantener el efecto del procedimiento secuencial debemos aplicar cada una de estas sustituciones al resto de los átomos del resolvente. Teniendo en cuenta que la operación de composición de sustituciones cumple:

$$(E\theta)\eta = E(\theta\eta)$$

podemos componer en primer lugar las sustituciones obtenidas y aplicarlas después al resolvente. Sin embargo el orden de aplicación de las sustituciones a variado respecto al secuencial y al no ser la composición de sustituciones conmutativa el resultado obtenido puede ser distinto del secuencial. Por lo tanto no siempre es posible explotar el paralelismo_Y y conservar la

semántica de Prolog. En general podemos asegurar que si θ es la sustitución computada hasta el momento y g_i y g_j son átomos, V es el conjunto de las varibles $x_i \in g_i\theta$ y U el conjunto de variables $y_i \in g_j\theta$ y se comple

$$U \cap V = \emptyset \tag{1.1}$$

entonces los átomos g_i y g_j pueden ser ejecutados simultaneamente sin que se altere la sustitución de respuesta respecto a la ejecución secuencial. Es decir, si las variables de un subconjunto de átomos de un resolvente estan ligadas a términos cerrados o independientes la ejecución simultanea es posible. Al conjunto de átomos del cuerpo de una cláusula que cumplen la condición de independencia y por tanto pueden ser ejecutados simultaneamente, lo denominamos *llamada paralela* y a cada uno de los átomos *objetivo independiente.*

Apliquemos el procedimiento de resolución al ejemplo de la derivada simbólica: $d(U, X, V)$ se cumple si V es la derivada de U respecto a X. Y $d2(U, X, V)$ representa a la derivada segunda.

$$\begin{array}{lll}
d(X, X, 1) & :- & var(X). \\
d(T, X, 0) & :- & atomic(T). \\
 & & \\
d(U + V, X, Du + Dv) & :- & d(U, X, Du), d(V, X, Dv). \\
d(U - V, X, Du - Dv) & :- & d(U, X, Du), d(V, X, Dv). \\
d(-(T), X, -(R)) & :- & d(T, X, R). \\
d(k * U, X, k * W) & :- & number(k), d(U, X, W). \\
d(U * V, X, b * U + a * V) & :- & d(U, X, a), d(V, X, b). \\
d(U/V, X, W) & :- & d(U * V^{-1}, X, W). \\
d(U^V, X, V * W * U^(V - 1)) & :- & number(V), d(U, X, W). \\
d(U^V, X, z * log(U) * U^V + V * W * U^(V - 1)) & :- & d(U, X, W), d(V, X, z). \\
d(log(T), X, R * T^{-1}) & :- & d(T, X, R). \\
d(exp(T), X, R * exp(T)) & :- & d(T, X, R). \\
d(sin(T), X, R * cos(T)) & :- & d(T, X, R). \\
d(cos(T), X, -(R) * sin(T)) & :- & d(T, X, R). \\
d(tan(T), X, W) & :- & d(sin(T)/cos(T), X, W). \\
 & & \\
d2(X, Y, V) & :- & d(X, Y, U), d(U, Y, V).
\end{array}$$

Dado el objetivo

$$d2(X + 1, X, Y)$$

Si explotamos el paralelismo_Y en los puntos de la derivación en que es posible, el proceso tiene la siguiente forma:
Resolvente:
$: -d2(X+1, X, y)$
cláusula aplicada: $d2(X_1, Y_1, Ddx)$
sustitución: $\{X/Y_1, X_1/Y_1+1, Y/Ddx\}$

$: -d(Y_1+1, Y_1, Dx), d(Dx, Y_1, Ddx)$
(no se puede aplicar paralelismo por que Dx se instancia a distintos valores)
cláusula aplicada: $d(U+V, X, Du+Dv)$
sustitución: $\{Y_1/U, V/1, X/U, Dx/Du+Dv\}$
$: -d(U, U, Du), d(1, U, Dv)$

(aplicamos ahora paralelismo. Analizando el programa sabemos que U no va a ser instanciada al unificar con ninguna cláusula y por lo tanto los átomos cumplen la condición de independencia)

Resolvente:
$: -d(U, U, Du)$
cláusula aplicada: $d(X_1, X_1, 1)$
sustitución: $\{U/X_1, Du/1\}$

Resolvente:
$: -d(1, U, Dv)$
cláusula aplicada: $d(T_1, X_1, 1)$
sustitución: $\{U/T_1, Dv/0\}$

(volvemos a la ejecución secuencial)
$: -d(1+0, U, ddx)$
cláusula aplicada: $d(T_2, X_2, 0)$
sustitución: $\{T_2/1+0, U/X_2, ddx/0\}$
□

La sustitución de respuesta computada es $\{X/y_1, Y/0\}$, que coincide con la secuencial.

1.4.2 Modelos de Paralelismo_Y

El principal problema para la implementación del paralelismo_Y es la detección de la dependencia entre los objetivos. Algunos sistemas explotan el *paralelismo_Y dependiente* como lo lenguajes lógicos concurrentes. Estos lenguajes tienen una sintaxis basada en cláusulas de Horn y por tanto similar a Prolog, pero presentan un modelo de operación más cercano a la programación procedural. Los principales representantes son PARLOG [18],

Concurent Prolog [59] y GHC (*Guarded Horn Clauses*) [67]. Los programas de estos lenguajes son conjuntos de cláusulas de Horn de la forma:

$$H : -G_1, ..., G_N | B_1, ..., B_m.$$

El operador *commit* | separa la *guarda* del cuerpo de la cláusula. La función de esta guarda es selecionar la cláusula para la resolución del predicado. Una vez seleccionada la cláusula no hay forma de deshacer la elección. El paralelismo que explotan estos sistemas es principalmente de tipo Y, pudiendo aparecer paralelismo_O durante la evaluación de la guarda. La mayoría de estos sistemas usan un modelo de ejecución basado en procesos de objetivos. Un proceso de objetivo es el encargado de probar cada cláusula candidata hasta encontrar una que encaje satisfactoriamente y entonces crear un proceso de objetivo para cada objetivo del cuerpo de la cláusula. Los intentos de unificación de los procesos de objetivo con el cuerpo de una cláusula pueden fallar, tener éxito o quedar suspendidos a la espera de la instanciación de una variable. Por lo tanto, estos sistemas necesitan crear, suspender, rearrancar y matar un número posiblemente alto de procesos de grano fino.

El sistema ANDORRA [72], propuesto por D.H. Warren [72] y que ha dado lugar a una variedad de extensiones desarrolladas por diferentes grupos de investigación, es otro sistema que explota el paralelismo_Y dependiente. Se trata de un modelo de ejecución de Prolog que integra Prolog con los *lenguaje lógicos concurrentes* haciendo del determinismo la base de la ejecución transparente del paralelismo_Y dependiente y del paralelismo_O. En este modelo, los objetivos deterministas (existe como máximo una cláusula candidata que encaje) se ejecutan con paralelismo_Y mientras hay objetivos deterministas a la izquierda. El paralelismo_O aparece si los objetivos restantes son indeterministas. La idea central es ejecutar determinados objetivos en primer lugar y en paralelo, retrasando la ejecución de los objetivos no deterministas hasta que los objetivos deterministas no pueden continuar.

Otros sistemas sólo explotan el paralelismo_Y de aquellos objetivos que no van a vincular la misma variable lógica. Se han propuesto distintos métodos para detectar la independencia de los objetivos:

- Método estático [16]
  Se basa en el análisis del programa en tiempo de compilación e indica el paralelismo que resulta de considerar el caso peor de dependencia entre objetivos. Con este esquema pueden perderse oportunidades

importantes de paralelismo

- Método dinámico [21]
  La dependencia del flujo de datos entre objetivos se detecta en tiempo de ejecución y se establece una relación productor-consumidor entre los objetivos. Se crea un grafo de flujo de datos que especifica el orden en que se resuelven los objetivos, no permitiendo la ejecución paralela de dos objetivos que pueden instanciar una misma variable. Con este método se detecta una gran cantidad de paralelismo, pero las comprobaciones en tiempo de ejecución pueden suponer un gran recargo.

- Método híbrido[28] [33]
  Los análisis en tiempo de compilación se combinan con test simples en tiempo de ejecución, explotandose lo que se denomina *Paralelismo_Y Restringido*.

Para explotar el paralelismo_Y Restringido los programas son compilados en grafos llamados *Expresiones de Grafo Condicionales* (CGEs), que en tiempo de ejecución únicamente requieren comprobaciones sobre la cerrazón e independencia de las variables. Por ejemplo, para la cláuslula

```
g(X) :- p(X),q(X).
```

se produciría la siguiente CGE:

```
g(X) = (ground(X) p(X) & q(X))
```

indicando que si X es una variable cerrada entonces los objetivos p y q se ejecutan en paralelelo y sino secuencialmente.

El sistema &-Prolog [35], con su máquina de ejecución paralela [33] se basa en este modelo e incorpora las instrucciones necesarias para la ejecución paralela. El modelo abstracto de memoria supone sistemas con memoria compartida y se basa en el uso de pilas, extendiendo la WAM mediante la incorporación de una pila de objetivos para la distribución de trabajo y las estructuras necesarias para el control de las ejecuciones paralelas y del mecanismo de backtracking. El modelo extiende el mecanismo de backtracking sobre las CGE's. Existen varios casos dependiendo de si el siguiente punto de

backtracking está dentro o más allá de una CGE o pertenece a una ejecución secuencial:

- Backtracking dentro de una CGE
  Puesto que los objetivos son independientes, las vinculaciones producidas por los objetivos distintos del que ha fallado no han causado el fallo. Por lo tanto, la evaluación paralela de la CGE debe abortarse y el backtracking continua en el punto que precede a la CGE.

- Backtracking sobre una CGE
  Normalmente, durante la ejecución de un CGE quedan alternativas sin explorar en los objetivos. El backtracking en la CGE se realiza de izquierda a derecha. El modelo requiere información sobre las alternativas pendientes de cada objetivo, para conducir el backtracking en el orden correcto.

La PWAM implementa el paralelismo_Y restringido manteniendo el mismo manejo de memoria y diseño de la máquina abstrata que la WAM. Verden [69] recoge estas ideas en un modelo productor-consumidor implementado sobre un sistema distribuido y añade un mecanismo de backtracking inteligente.

1.4.3 Paralelismo_O

Aplicar paralelismo_O a la resolución de un objetivo significa unificar simultaneamente un átomo A seleccionado de un resolvente con un conjunto de cláusulas $A_i, ..., A_j$ del programa (***cláusulas paralelas***). Cada una de estas unificaciones producirá una sustitución θ_i y un conjunto de átomos que deben reemplazar al átomo que se seleccionó. Si estas operaciones se realizan sobre un mismo resolvente se produce una vinculación multiple de sus variables, pués todas se refieren a la unificación de un mismo átomo.

Las cláusulas alternativas que se han tomado en computaciones paralelas no pueden aparecer en los puntos de elección asociados a A_i en cada computación. Además si hay cláusulas que unifican con A_i que no han sido elegidas en ninguna de las computaciones paralelas no pueden aparecer en más de uno de los puntos de elección asociados a A_i. Es decir, si PE_i y PE_j son puntos de elección asociados a A_i en distintas computaciones entonces

$$PE_i \cap PE_j = \emptyset$$

Para estudiar este tipo de paralelismo nos apoyaremos en el conocido programa de *Reinas*, que busca las posibles colocaciones de N reinas en un tablero $N \times N$ de manera que ninguna esté amenazada.

$safe(N, Queens)$ $:-$ $extend(0, N, [], Queens)$.

$extend(N, N, Qs, Qs)$.
$extend(M, N, Selected, Queens)$ $:-$
$M1 = M + 1$,
$choose(1, N, C)$,
$consistent(q(M1, C), Selected)$,
$extend(M1, N, [q(M1, C)|Selected], Queens)$.

$choose(M, N, M)$.
$choose(M, N, K)$ $:-$ $M < N$,
$M1 = M + 1$,
$:-$ $choose(M1, N, K)$.

$consistent(Q, [])$.
$consistent(Q, [Q1|Rest])$ $:-$ $no\_attack(Q, Q1)$,
$consistent(Q, Rest)$.

$no\_attack(q(X1, Y1), q(X2, Y2))$ $:-$ $different(Y1, Y2)$,
$D = ||X1 - X2|| - ||Y1 - Y2||$,
$D = / = 0$.

Tomaremos N = 4, siendo el objetivo:

$$:- safe(4, X).$$

En este ejemplo podemos considerar el paralelismo del predicado $choose(1, N, C)$ que para $N = 4$ proporciona los valores de C $\{1, 2, 3, 4\}$. La Figura 1.6 presenta un esquema del árbol de búsqueda de este programa. Vemos que cada elección distinta que hacemos entre las cláusulas del predicado *choose* lleva a un nodo terminal distinto, produciendose en cada uno de ellos distintas vinculaciones sobre las mismas variables.

1.4.4 Modelos de paralelismo_O

La implementación del paralelismo_O necesita resolver distintos problemas dependiendo del tipo de sistema elegido para la explotación. En teoría, se genera un nuevo resolvente en cada unificación de un objetivo con la cabeza de una cláusula, renombrando todas las variables del resolvente actual. En

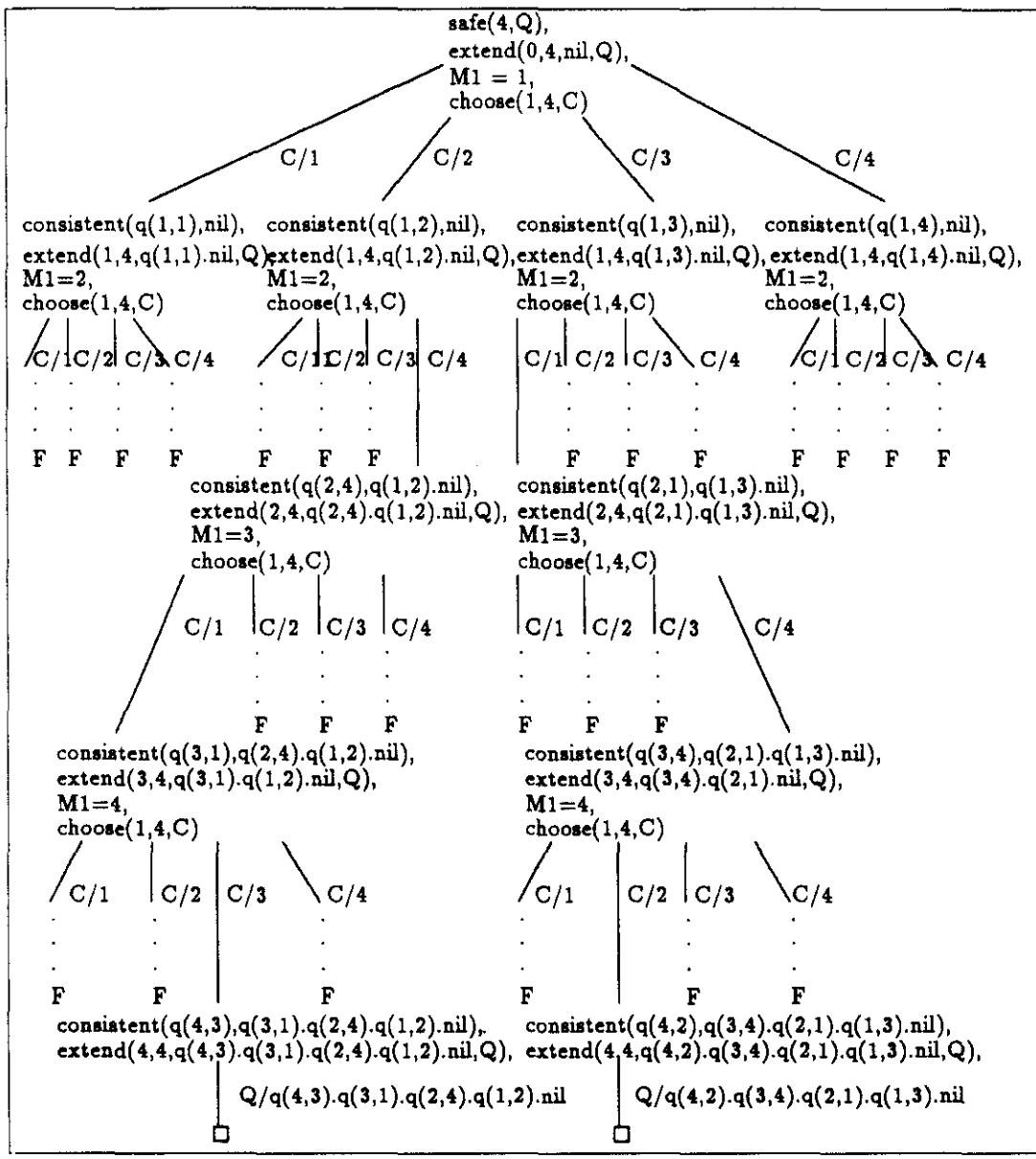


Figura 1.6: Arbol de búsqueda de queens

la práctica, en la mayoria de los sistemas con memoria compartida los resolventes no son replicados por razones de eficiencia y se utilizan las mismas posiciones de memoria para computar los resolventes alternativos.

En la implementación secuencial basada en la WAM, las variables se identifican por sus posiciones de memoria. En cada momento de la computación a cada posición de variable se le asocia una única variable. Cuando se produce backtracking las posiciones de memoria correspondientes a variables de los objetivos eliminados, se inicializan para ser reutilizadas. En los sistemas que explotan paralelismo_O, las computaciones paralelas equivalen a una serie de computaciones secuenciales encadenadas por operaciones de backtracking y por tanto distintos elementos de proceso pueden asociar la misma posición de memoria de la pila a diferentes variables lógicas. Las soluciones aportadas para resolver este problema pueden clasificarse en aquellas para sistemas que comparten las pilas y las de sistemas en los que cada elemento de proceso tiene asociado un espacio de memoria en exclusiva.

Pilas compartidas

En los sistemas lógicos paralelos que comparten pilas, se utilizan estructuras de datos especiales para almacenar las variables lógicas que pueden tener el mismo identificador en la pila que otras variables lógicas usadas por otros elementos de proceso. Se han propuesto dos tipos de estructuras de datos: *arrays o ventanas de vinculaciones* [71] y *hash-windows* [74].

Un array de vinculaciones de un elemento de proceso es un array privado, que contiene las variables lógicas que tienen la misma identificación que las pertenecientes a otros elementos de proceso. En lugar de contener un valor, la posición de la pila compartida contiene un indice. Este indice señala a diferentes variables lógicas, cada una en el array de vinculaciones de los elementos de proceso que comparten ese segmento de la pila. El sistema Aurora [15] está basado en el modelo SRI [73] que utiliza arrays de vinculaciones para almacenar las vinculaciones de las variables logicas. Las pilas de la WAM se extienden a pilas de *cactus*, que reflejan la forma del árbol de búsqueda. Los objetos de las pilas físicas de los elementos de proceso pueden linkarse, formando una pila lógica.

Otra solución [74] consiste en almacenar las variables que comparten la misma identificación en estructuras de datos llamadas *hash-windows* y asociarlas a las ramas del árbol de búsqueda. Una entrada a la *hash-window* se computa por "hashing" de la identificación (dirección en la pila) de la

variable compartida.

Pilas exclusivas

Cuando cada elemento de proceso en un sistema con paralelismo_O accede únicamente a sus propias pilas, es necesario que los elementos de proceso desocupados que consiguen trabajo tengan una réplica de las pilas del elemento que proporciona el trabajo. Los métodos para conseguir la replica de las pilas son la *copia* [2] y la *recomputación* [19] [9].

En el esquema de copia de pilas, un elemento de proceso desocupado que va a colaborar en la explotación del paralelismo_O, consigue una copia de los segmentos de las pilas que corresponden a la computación desde el nodo raiz hasta en nodo en que surge el paralelismo a explotar. El sistema Kabu-Wake [51] fué el primero en copiar las pilas al cambiar de tarea y en usar marcas de tiempo para desechar vinculaciones. En el modelo MUSE [2] los elementos de proceso activos, comparten con los desocupados una serie de puntos de elección que contienen las alternativas pendientes. Además en este sistema el recargo debido a la copia de las pilas se reduce aprovechando la parte del árbol de búsqueda que tengan en común los elementos de proceso que comparten el trabajo.

Siguiendo el esquema de recomputación, cada elemento de proceso computa una rama del árbol desde la raíz. Las diferencias entre los modelos existentes basados en recomputación, consisten en como se realiza en reparto de las ramas a explotar en paralelo entre los elementos de proceso disponibles. En [19] cada elemento de proceso computa un camino predeterminado del árbol de búsqueda bajo el control de un proceso especializado llamado "controlador", mientras que en [9] el camino a seguir se construye en la ejecución previa a la explotación del paralelismo.

1.4.5 Modelos de Combinación del paralelismo

La combinación del paralelismo_Y y el O presentan nuevas dificultades de implementación. La explotación del paralelismo del árbol de búsqueda completo puede generar una explosión combinatoria de trabajos que el sistema no pueda manejar, por lo que se suelen aplicar algunas restricciones. La combinación del paralelismo Y_restringido_O requiere la generación del producto cruzado o combinación de las distintas soluciones de los objetivos de la llamada paralela. El control de la ejecución debe garantizar que se realiza

el producto cruzado de todas las soluciones de las ramas_Y, sin introducir un excesivo recargo al almacenar las soluciones parciales o al sincronizar a los elementos de procesoes.

J.S. Conery [22] ha desarrollado un modelo de paralelismo_Y\O que explota paralelismo de grano fino. El modelo consiste en una colección de procesos_Y\O asíncronos que se comunican únicamente por medio de mensajes. Cada instanciación de una cláusula se controla por un proceso_Y cuya función es decidir que literales del cuerpo de la cláusula serán los siguientes en evaluarse. Cuando un proceso_Y ha seleccionado uno o más literales se crea un proceso_O para cada uno de ellos que se encarga de la resolución del literal. En este modelo se evaluan en paralelo solo aquellos átomos que no tienen variables en común o cuyas variables compartidas ya han sido vinculadas. El acceso a las variables compartidas por varios elementos de proceso se resuelve por el método denominado de *entornos cerrados*, siendo un entorno una colección de estructuras. Una estructura está cerrada si ninguna de sus variables se define en términos de variables que pertenecen a otras estructuras.

El sistema PPP (Parallel PROLOG Processor) del proyecto Aquarius mejora el rendimiento de Prolog combinando paralelismo_Y Restringido, paralelismo_O y backtracking inteligente. Se basa en un modelo de procesos_Y_O como el propuesto por Conery. Resuelve el problema de las vinculaciones multiples que se producen al explotar el paralelismo_O mediante *ventanas de vinculaciones*. Para resolver el problema del producto cruzado introduce dos conceptos: contención y encadenamiento dinámico de ventanas. La contención consiste en que los procesos_Y que ejecutan un objetivo de una llamada paralela no puedan crear procesos_O. En el modelo estandard de ventanas de vinculaciones el conjunto de vinculaciones visibles a un proceso está contenido en una cadena estática de ventanas, mientras que en PPP esta cadena es dinámica, enlazando y desenlazando ventanas de forma que se construyan los conjuntos de vinculaciones visibles a los procesos apropiados.

El modelo Reduce-OR [45] intenta explotar todo el paralelismo disponible en los programas lógicos. Para conseguirlo de forma eficiente, el modelo divide la computación en computaciones independientes, que se corresponden con los nodos del árbol de búsqueda y supone una computación de grano fino. La combinación del paralelismo_Y\O se resuelve haciendo distinciones en el algoritmo de unificación: los nodos_O del árbol siguen un algoritmo

basado en los entornos cerrados propuestos por Conery, mientras que para los nodos_Y el algoritmo recoge las vinculaciones producidas por diferentes llamadas a un objetivo.

El sistema PEPSys [74] introduce un algoritmo de control basado en la producción perezosa de las soluciones a combinar, de manera que no se pierdan por backtracking soluciones que aun no han sido combinadas con todas las de los restantes objetivos de una llamada paralela. El sistema utiliza un esquema de *hash-windows* para el manejo de las vinculaiones multiples de las variables. El producto cruzado se forma emparejando *hash-windows* a las diferentes ramas Y_paralelas de forma que se representen todos los grupos de soluciones parciales.

El sistema LORAP [13] (*Limited OR-Parallelism and Restricted AND-Parallelism*) presenta un esquema de explotación del paralelismo combinado dirigido por demanda o perezoso. Una variable no se vincula hasta que el objetivo es probado. El modelo de ejecución se basa en la generación de procesos. Cada cláusula corresponde a un proceso que a su vez consta de uno de ejecución hacia delante y otro de backtracking.

El modelo dirigido por datos [65] se basa en seguir el flujo de datos para la explotación del paralelismo combinado. La máquina abstracta correspondiente es una máquina de flujo de datos que permite activaciones multiples del mismo grafo del flujo de datos. Un nodo del grafo se activa cuando todas sus entradas estan disponibles. Las entradas disponibles se distinguen utilizando etiquetas o colores.

El modelo ACE [30] combina el paralelismo_Y independiente con el modelo de paralelismo_O del sistema MUSE de manera que ambos sistemas se mantienen sin apenas modificaciones.

2

P.D.P.: Procesador Distribuido de Prolog

2.1 Introducción

PDP (Procesador Distribuido de Prolog) es un sistema multisecuencial (cada procesador tiene asociado una única computación) para la explotación del paralelismo_Y_Restringido\O de Prolog. El sistema está soportado por una arquitectura distribuida con un alto número de procesadores que trabajan bajo un control jerárquico (Figura 2.1). Con objeto de reducir el número de mensajes intercambiados, el control se realiza en una serie de procesadores llamados *controladores*. El resto de los procesadores, *procesadores básicos*, están dedicados a la ejecución de programas Prolog. Cada controlador está asignado a un grupo de procesadores básicos. Si el número de procesadores es suficientemente elevado, los controladores asociados a cada grupo centralizan su información en un controlador de nivel superior. La ejecución comienza en un procesador básico configurado como *inicial*. Los procesadores básicos desocupados piden trabajo al controlador, que toma nota de estas peticiones. Cuando un procesador básico activo encuentra paralelismo informa de ello al controlador, que le indica a cual de los procesadores desocupados debe enviar el trabajo pendiente.

El modelo de ejecución de PDP explota paralelismo_Y formando *tareas_Y* autónomas para la computación de cada objetivo independiente. La tarea padre espera la respuesta a cada una de estas tareas_Y. Para el control de la ejecución de las tareas paralelas se utiliza una adaptación a sistemas distribuidos del esquema desarrollado por Hermenegildo [33] para sistema

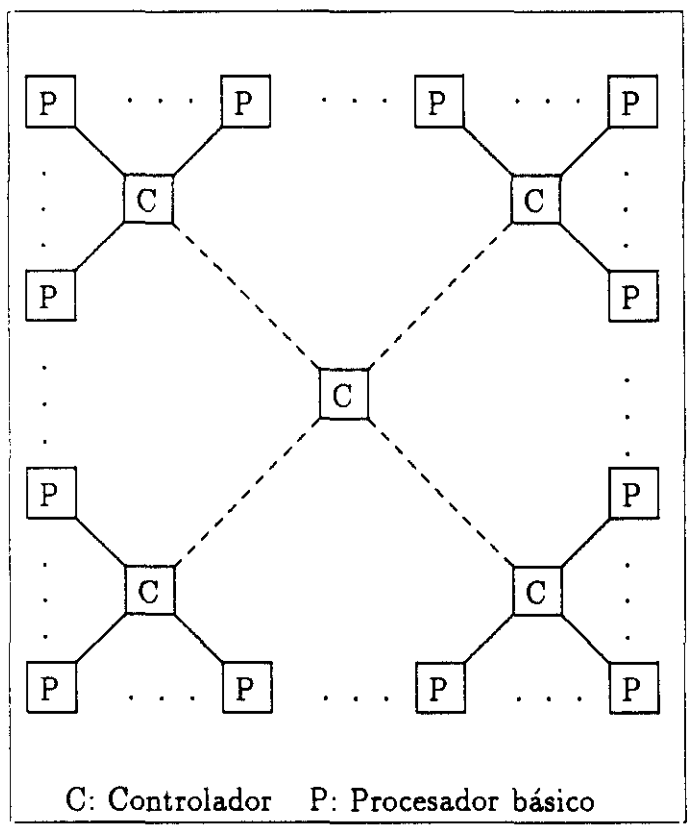


Figura 2.1: Organización del sistema

de memoria compartida. La explotación del paralelismo_O se basa en la ejecución multisecuencial de las ramas del árbol de búsqueda. Cuando una tarea encuentra una cláusula paralela, mediante un aviso al controlador de su grupo pone a disposición de los procesadores desocupados la *tarea_O* correspondiente. Puesto que los procesadores trabajan en entornos cerrados, cuando se crea una tarea_O se reconstruye previamente el entorno de la tarea padre. Para ello, el procesador *recomputa* el objetivo inicial siguiendo el *camino de éxito* (sin backtracking) recibido de la tarea padre.

Cuando el paralelismo aparece combinado en forma de Y_bajo_O se explota de la misma forma que en el caso del paralelismo_Y puro, ya que la explotación del paralelismo_O reproduce el estado del entorno de trabajo al que se llegaría con una ejecución secuencial. En cambio, cuando un programa presenta paralelismo O_bajo_Y, se tienen que combinar las distintas soluciones de los objetivos paralelos. PDP evita almacenar soluciones parciales y sincronizar tareas produciendo la combinación de forma distribuída, es decir, no hay una tarea encargada de realizar el producto cruzado de las distintas alternativas, sino que cada nueva tarea que se crea conoce automaticamente que combinaciones de soluciones le corresponden. La idea es crear una computación para cada combinación de soluciones, recomputando el camino de éxito que lleva del objetivo inicial a la llamada paralela considerada. De esta forma la explotación del paralelismo_Y se realiza con el mecanismo del paralelismo_O, creando tareas autónomas que reducen considerablemente el tráfico de mensajes.

Para describir gráficamente el modelo de ejecución de PDP utilizaremos un *árbol de tareas*, es decir un árbol cuyos nodos pueden ser de dos tipos:

- **Tarea_O:** Corresponde a la ejecución del objetivo inicial siguiendo un camino determinado del árbol de búsqueda.

- **Tarea_Y:** Corresponde a la ejecución de un objetivo paralelo con la instanciación de sus variables. La respuesta obtenida es enviada a la tarea padre de la que se recibió el trabajo.

En general, el árbol de tareas y el árbol de búsqueda son diferentes, ya que no siempre es posible ni conveniente explotar todo el paralelismo potencial de programa.

2.2 Explotación del Paralelismo_Y

PDP explota paralelismo_Y creando tareas_Y autónomas para la computación de cada objetivo de una llamada paralela. La Figura 2.2 representa la ejecución de una llamada paralela. La tarea padre, que puede ser de tipo Y u O, crea tantas *tareas_Y* como objetivos independientes hay en la llamada paralela. Cuando finaliza la ejecución de una tarea_Y, la respuesta es enviada a la tarea padre, que sólo puede continuar la ejecución cuando ha recibido todas las respuestas correspondientes a los objetivos paralelos.

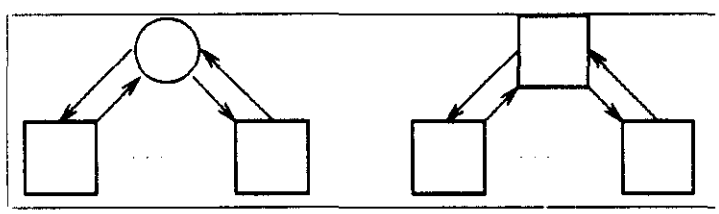


Figura 2.2: Explotación del paralelismo_Y

Las *tareas_Y* reciben la instanciación de las variables del objetivo que se les asigna para poder realiza la computación de forma autónoma. El ejemplo de la Figura 2.3 ilustra el proceso para el siguiente programa:

```
Programa:
P1:    p(a,a2).
P2:    p(b,b2).
P3:    p(f(X,Y), f(XX,YY) :- p(X,XX),p(Y,YY).

Objetivo:    :- p(f(f(a,a),f(b,b))), X).
```

La tarea 2 recibe el objetivo $p(Y, YY)$, junto con la instanciación:

$$Y \leftarrow f(b, b)$$

de manera que puede realizar la computación autónomamente. Cuando finaliza envía a la tarea padre la instanciación de las variables libres del objetivo recibido

$$YY \leftarrow f(b2, b2)$$

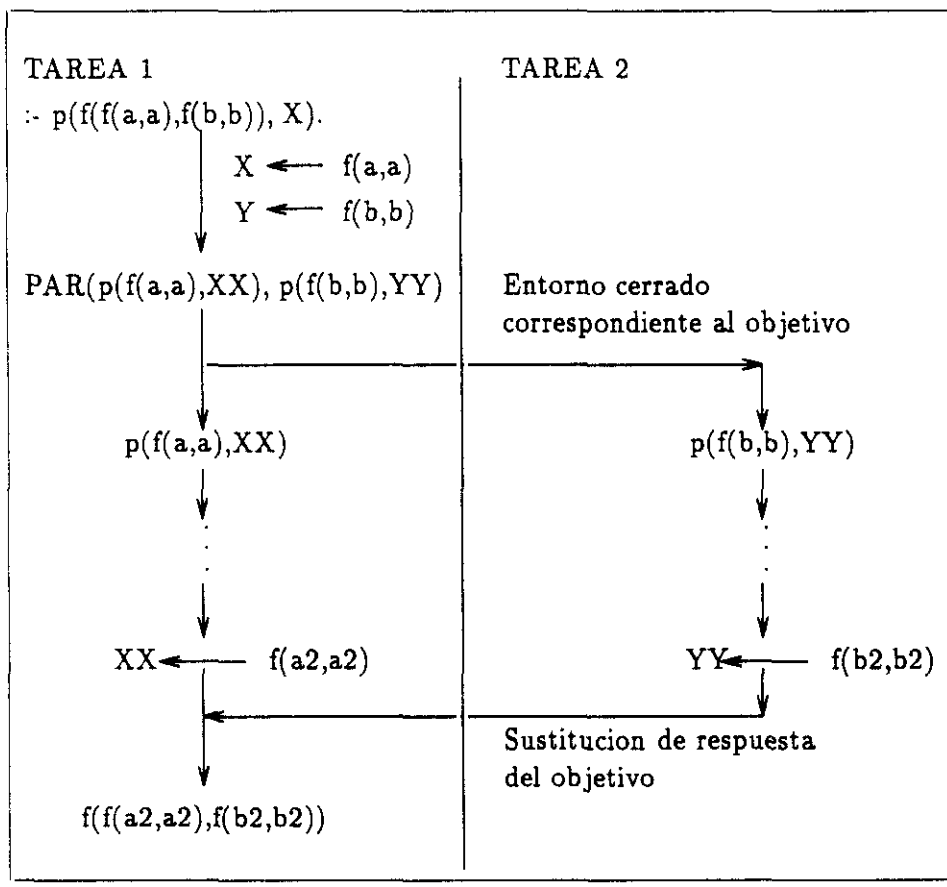


Figura 2.3: Ejemplo de explotación del paralelismo_Y

Cuando la tarea padre recibe la respuesta, vincula las variables del objetivo a la instanciació recibida y si ha terminado de ejecutar el resto de los objetivos de la llamada, continua la ejecución.

El mecanismo de explotación del paralelismo_Y se presenta de forma precisa en la Figura 2.4. El *entorno* de ejecución está constituido por el *resolvente* R, la *sustitución de respuesta* V, y la *pila de backtracking* S que almacena los sucesivos estados del resolvente y la sustitución. El procedimiento de ejecución con paralelismo_Y consiste en un ciclo de pasos de resolución que se repite hasta que el resolvente está vacio (ejecució con éxito) o se produce un fallo. Existen dos tipos de pasos de resolución, dependiendo de la existencia de paralelismo_Y. Si a la izquierda del resolvente existe un conjunto independiente de objetivos (*conjunto_independiente*) se explota el paralelismo_Y creando una tarea_Y paralela para la ejecución de cada uno de ellos. Cada una de estas tareas recibe el programa, el objetivo asignado y la vinculación de las variables del objetivo, es decir, la restricción de la sustitución computada hasta el momento restringida a las variables del objetivo (*restriccion*(V, A_i)). La computación de estas tareas produce un resultado ($result_i$) de éxito o fallo. Si el resultado es de éxito, la tarea_Y proporciona la sustitución de respuesta computada correspondiente al objetivo (θ_i). Si el resultado de todas la tareas_Y es de éxito, para cada una de ellas se aplica al resolvente pendiente la sustitución de respuesta computada (R := aplicar(θ_i,R)) y se compone dicha sustitución con la general($v := v \circ \theta$). Si todos los objetivos de la llamada paralela terminan con éxito, la función *completa* lo anota para tenerlo en cuenta en caso de backtracking. El otro tipo de paso de resolución se produce cuando no existen objetivos independientes a la izquierda del resolvente y por tanto se ejecuta un único objetivo. En este caso la ejecución es de la misma forma que en el caso secuencial, es decir, se busca el procedimiento asociado al átomo a resolver, intentando la unificación con cada una de sus cláusulas hasta obtener éxito o haber comprobado todo el procedimiento. Si el resultado es de éxito se almacena en la pila de backtracking la parte del procedimiento que ha quedado sin explorar P_a, el resolvente R y la sustitución de respuesta V, se compone la sustitución obtenida con la general y se reemplaza en el resolvente el átomo unificado por el cuerpo de la cláusula con la que se ha unificado. Si al dar un paso de resolución de cualquiera de los dos tipos se produce un resultado de fallo se realiza *backtracking* recuperando un estado anterior del entorno de la pila de backtracking.

```
procedure ejecucion_Y(p:programa; R:resolvente; S:pila de backtracking;
            V:sustitucion; resultado:tipo_res);
var
    a : objetivo;      P_a: procedimiento asociado al predicado a;
    c : clausula;      V' : sustitución;      r,i : integer;
    θ: array[1..NMAX] of sustitucion;
    result: array[1..NMAX] of tipo_res;
begin
   repeat
       [A_1, ..., A_r] := conjunto_independiente(R);
       if r > 1 then begin
          parbegin /* PARALELISMO Y */
             tarea_Y(p,A_1,restriccion(V,(A_1)),result_1, θ_1);
                                  .
                                  .
                                  .
             tarea_Y(p,A_r,restriccion(V,(A_r)),result_r, θ_r);
          parend
          i := 1;
          while (i <= r) and resultado = exito do begin
             resultado := result_i;
             if resultado = exito then begin
                R := aplicar(θ_i,R);
                V := V ∘ θ_i;
             end
          end
          if resultado = exito then completa(L);
       end
       else begin /* r = 1 */
          a := R[1] /* Se toma el primer atomo del resolvente */
          buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
          repeat
             c := P_a[1]; /* Se toma la primera clausula del procedimiento */
             P_a := P_a - c;
             resultado := unificar(a,c,V');
          until (P_a = []) or resultado = exito;
          if resultado = exito then begin
             if P_a <> [ ] then salvar(P_a,R,V,S);
             V := componer(V,V');
             reemplazar(R,a,c); /* En R, reemplazar a por el cuerpo de c */
          end
       end
       else if resultado = fallo and not vacio(S) then
          backtracking(p,S,R,V,resultado);
       else resultado := fallo
    until (R = []) or resultado = fallo;
end
```

Figura 2.4: Algoritmo de explotación del paralelismo_Y

```
proccess tarea_Y(p:programa;Q:objetivo; V:sustitucion;
                 resultado:tipo_res; V2: sustitucion);
begin
    R = [Q];
    S = ∅
    ejecucion(p,R,S,V,resultado);
    if resultado = exito then
       V2 := restriccion(V,Q);
end
```

Figura 2.5: Esquema de creación de una tarea_Y

La explotación multisecuencial del paralelismo_Y en nuestro sistema requiere que las *tareas_Y* dispongan de toda la información necesaria para la resolución del objetivo paralelo recibido, esto es, las instanciaciones producidas sobre las variables del mismo. La creación de una *tarea_Y* se presenta en la Figura 2.5. El proceso recibe el objetivo a ejecutar junto con la restricción a las variables del objetivo de la sustitución de respuesta producida hasta el momento que produce la función $sust(var(Q))$. Una vez ejecutado el objetivo se obtiene la sustitución de respuesta computada, restringida a las variables del objetivo recibido.

2.2.1 Backtracking

El procedimiento de backtracking de la ejecución secuencial se extiende para recoger los nuevos casos que se presentan debido a la explotación del paralelismo_Y. Es necesario considerar los fallos que no se producen en la propia tarea sino en las tareas_Y creadas por ella. Además se aplican técnicas de *backtracking inteligente* (BI): no se exploran exhustivamente las alternativas pendientes de las que se sabe que no llevarán a una solución. En PDP estas técnicas se aplican en dos situaciones, cuando falla uno de los objetivos de una llamada paralela en curso, y cuando se requiere la reevaluación de un objetivo ejecutado por otra tarea. Puesto que un objetivo de una llamada paralela es independiente de los restantes objetivos de la llamada, las distintas soluciones de cada uno de ellos no alterarán un resultado de fallo y por tanto, se considera que falla la llamada completa. Mecanismos más sofisticados de BI tratan los fallos de forma que sólo se retrocede al anterior punto

alternativo que ha participado en las instanciaciones que han dado lugar al fallo, como los presentados por Codognet[20] o Lin[48]. Estos mecanismos penalizan la ejecución hacia adelante y hacia atrás, por lo que sólo en programas con mucho indeterminismo resulta rentable su aplicación. Sin embargo, no todos los programas con alto indeterminismo consiguen mejorar su eficiencia con BI como señaló Fagin[29]. Estas consideraciones nos han llevado a aplicar técnicas de BI sólo para evitar reevaluaciones remotas de los objetivos que no van a arreglar el fallo producido. Este mecanismo de BI consiste en marcar las variables que no se han instanciado en la propia tarea, es decir cuya instanciación se ha recibido en la sustitución de respuesta computada por una tarea_Y, mediante la función *marcada(X, objetivo$_i$)* que asocia a la variable X el objetivo i que la instanció. Cuando al ejecutar un objetivo se accede a las varibles marcadas, se anota que exite una relación entre el objetivo paralelo que instanció la variable y el que se está computando:

if marcada(X, *objetivo$_i$*) **then**
  relacionado(*objetivo$_i$*);

Cuando se produce un fallo y se necesita la reevaluación de un objetivo paralelo ejecutado por otra tarea, se comprueba que el objetivo que ha fallado está *relacionado* con el objetivo paralelo a reevaluar, pués en caso contrario la nueva respuesta no alteraría el fallo.

Dependiendo del estado de la llamada paralela los casos posibles de fallo son los siguientes:

- *La llamada paralela no ha finalizado:*
  Se produce fallo de la llamada paralela completa, interrumpiendo las tareas_Y pendientes correspondientes a otros objetivos paralelos de la llamada y se busca una solución alternativa en un resolvente anterior.

- *La llamada paralela se ha completado con éxito:*

  - *No hay objetivos paralelos con cláusulas alternativas pendientes de explorar:*
    Se retrocede hasta el objetivo anterior a la llamada.

  - *Hay un objetivo paralelo ejecutado por la propia tarea con alternativas pendientes:*
    Se busca la siguiente solución de este objetivo.

- *Hay un objetivo paralelo ejecutado por una tarea_Y hija con alternativas pendientes:*
  - * *Cumple la condiciones de BI: objetivo relaccionado*
    Se pide una nueva solución a la tarea_Y que ejecutó el objetivo
  - * *No cumple la condiciones de BI: objetivo no relaccionado*
    Se busca otro objetivo de la llamada con alternativas pendientes.

El algoritmo de backtracking se presenta en la Figura 3.18. Se recupera de la pila de backtracking el objetivo a reevaluar, comprobandose si pertenece a una llamada paralela (*en_llamada_paralela*), lo que está anotado en el objetivo. Si está en una llamada paralela se comprueba si se había completado su ejecución. Una llamada paralela incompleta falla globalmente al fallar uno de sus objetivos, y por lo tanto se interrumpen las tareas_Y encargadas de la ejecución de objetivos de la llamada que aún estuviesen pendientes. En una llamada completa se busca un objetivo con alternativas pendientes. Si el objetivo se ha ejecutado por otra tarea, solo se pide su reevaluación si está *relacionado* con el objetivo que ha fallado. En otro caso se continua haciendo backtracking. Si el objetivo es local, se *reevalua* como se describe en el Capítulo 1.

2.3 Explotación del Paralelismo_O

Cuando se explota paralelismo_O se intenta simultaneamente la unificación del objetivo a resolver con las distintas cláusulas del procedimiento asociado. Aunque no hay restricciones teóricas a la explotación de este tipo de paralelismo, por razones de eficiencia sólo se debe explotar en aquellas ramas del árbol de búsqueda con grano suficiente para compensar el retardo debido al mecanismo paralelo.

Una tarea que encuentra paralelismo_O crea una *tarea_O* que resuelve el objetivo inicial explorando una determinada parte del árbol de búsqueda. La solución obtenida por la nueva tarea es enviada directamente a la entrada\salida, sin notificar nada a la tarea padre (Figura 2.7). La creación de una tarea_O en un sistema multisecuencial, en el que cada tarea accede únicamente a su propio entorno de trabajo, requiere la reconstrucción del entorno de la tarea padre. Para realizar esta reconstrucción se han considerado los dos métodos

```
procedure backtracking(p:programa; S: pila de backtracking;
                R:resolvente; V:sustitución; resultado:tipo_res);
begin
    if not vacio(S) then
        sacar(R,S); a := R[1];
        sacar(P_a,S); sacar(V,S);
        if en_llamada_paralela(a,L) then
        begin
            if llamada_incompleta(L) then begin
                eliminar_tareas(Llamada_paralella);
                resultado := backtracking(p,S,R,V);
            end
            else /* llamada paralela completa */
            begin
                incompleta(L);
                if remoto(a) and relacionado(a) then begin
                    resultado := pedir_reevaluacion(tareaY_a, a, θ_a);
                    R := aplicar(θ_a,R)
                    V := V ∘ θ_a
                end
                else if not remoto then
                    resultado := reevaluacion(p, a, P_a, R, S, V);
                if resultado = exito then completa(L);
            end
        end
        else /* no pertenece a una llamada paralela */
            resultado := reevaluacion(p, a, P_a, S, R , V);
    end
    else resultado := fallo;
end
```

Figura 2.6: Algorítmo de backtracking

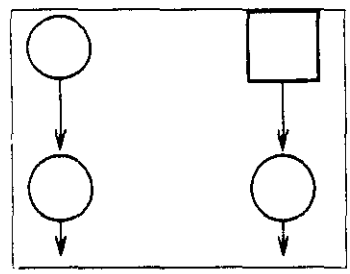


Figura 2.7: Explotación del paralelismo_O

de pilas exclusivas presentados en el Capítulo 1: copiar el entorno de trabajo y recomputar el objetivo inicial hasta reconstruir el entorno.

2.3.1 Reconstrucción del entorno por copia

```
proccess tareaO_copia(p:programa; R:resolvente;
                      S:pila de backtracking; V:sustitucion);
var
    Q: objetivo;
    resultado:tipo_res;
begin
    Q = R[1];
    ejecucion(p,R,S,V,resultado);
    if resultado = exito then begin
        V := restricción(V,Q);
        visualizar(V);
    end
end
```

Figura 2.8: Creación de una tarea_O por copia

Una *tareaO_copia* se crea a partir de una copia del entorno de trabajo de la tarea padre. El algoritmo de creación de una tarea_O se describe por un proceso cuyos parametros representan la información enviada por la tarea padre (Figura 2.8). La nueva tarea parte del entorno de trabajo de la tarea padre (recibe la pila de backtracking, el resolvente y la sustitución de respuesta computada hasta el momento) y hace *backtracking* para tomar la alternativa siguiente a la explorada por la tarea padre. A partir de ese

momento, ambas tareas exploran diferentes caminos del árbol de búsqueda. La cantidad de información transmitida al proceso *tarea_copia* puede ser muy grande dependiendo del programa y de los datos de entrada.

El algoritmo de ejecución que contempla la explotación del paralelismo_O por este método aparece en la Figura 2.9. Al igual que en el caso secuencial, el resolvente se transforma hasta estar vacio (ejecución con éxito) o hasta que es imposible realizar más transformaciones por estar vacia la pila de backtracking en la que se acumulan las alternativas pendientes de probar (ejecución con fallo). La diferencia se encuentra en que al considerar cada cláusula del procedimiento correspondiente al objetivo, se comprueba si está marcada para ser ejecutada en paralelo (*paralelismoO(cláusula)*), y si es así se crea una nueva tarea_O (*tareaO_copia*) que explora la siguiente cláusula del procedimiento P_a asociado al objetivo.

2.3.2 Reconstrucción del entorno por Recomputación

Otra posibilidad de reconstruir un estado de resolución es volver a ejecutar el objetivo inicial, es decir, *recomputarlo*. La nueva ejecución se hace determinista utilizando los datos de la tarea padre para evitar las alternativas que llevan a un fallo, es decir sigue el *camino de éxito* de la tarea padre: en la Figura 2.10, $(C2, C5, C10)$ representa el camino de éxito, indicando que C4 y C9 ya han sido exploradas. La creación de una tarea por este método (Figura 2.11) consiste en *recomputar* el objetivo a resolver siguiendo el camino de éxito recibido hasta el punto en que apareció el paralelismo. En este punto el camino lleva a una alternativa diferente de la tomada por la tarea padre y continua la ejecución anotando su propio camino de éxito. Por lo tanto las ejecuciones que contemplen la explotación del paralelismo_O por este método deben registrar la alternativa tomada en cada paso de resolución, actualizando el camino de exito cada vez que se produzca backtracking. El procedimiento de ejecución se presenta en la Figura 2.12. Como en el caso secuencial, el procedimiento consiste en una secuencia de pasos de resolución que se repite hasta que el resolvente queda vacio o se produce fallo. En cada paso de resolución se toma un objetivo del resolvente y se busca el procedimiento asociado (*buscar_procedimiento*). Se intenta la unificación con cada una de las cláusulas del resolvente hasta tener éxito o haberlas probado todas. Para cada cláusula se comprueba si está marcada para la explotación del paralelismo, en cuyo caso se crea una nueva *tarea_O* que se encarga de la exploración de la parte del árbol de búsqueda

```
procedure ejecucion_O(p:programa; R:resolvente;
                S:pila de backtracking: V:sustitucion; resultado:tipo_res);
var
    P_a: procedimiento asociado al predicado a;
    resultado:tipo_res;
    a : objetivo;
    c : clausula;
    V' : sustitución;
begin
    backtracking(p,S,R,V);
   repeat
       a := R[1] /* Se toma el primer atomo del resolvente */
       buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
       repeat
          c := P_a[1]; /* Se toma la primera clausula del procedimiento */
          P_a := P_a - c;
          if paralelismoO(c) then
          begin
             tareaO_copia(p,R,S,V);
             P_a := P_a - P_a[1];
          end
          resultado := unificar(a,c,V');
       until (P_a = []) or resultado = exito;
       if resultado = exito then
       begin
          if P_a <> [ ] then salvar(P_a,R,V,S);
          V := componer(V,V');
          reemplazar(R,a,c); /* En R, reemplazar a por el cuerpo de c */
       end
       else if not vacio(S) then begin
          resultado := backtracking(S,a,P_a,R,V);
       end
       else resultado := fallo
    until (R = []) or resultado = fallo;
end
```

Figura 2.9: Algoritmo de explotación del paralelismo_O por copia

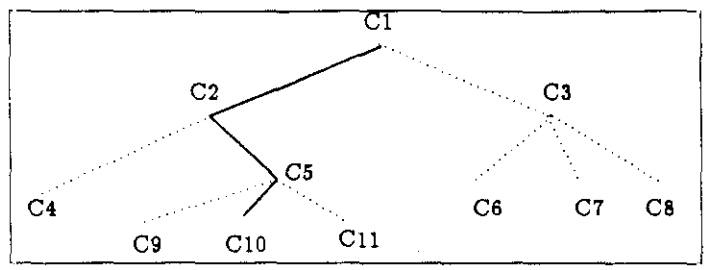


Figura 2.10: Camino de éxito

que se encuentra a partir de esta cláusula. Cuando la unificación tiene éxito, además de salvar el estado del entorno en la *pila de backtracking*, componer la sustitución obtenida con la general y reemplazar el objetivo por el cuerpo de la cláusula considerada en el resolvente, se salva en el *camino de éxito* la cláusula considerada (*meter(c,C)*).

```
proccess tareaO_recomp(p:programa;Q:objetivo:C:camino de exito);
var
    S : pila de backtracking;
    V : sustitucion;
    resultado:tipo_res;
begin
    S := ∅
    R := [Q]
    recomputar(p,R,C,V);
    ejecucion(p,R,S,V,resultado);
    if resultado = exito then begin
        V := restricción(V,Q);
        visualizar(V);
    end
end
```

Figura 2.11: Creación de una tarea_O por recomputacion

La recomputación (Figura 3.1) es un proceso determinista que sigue el camino de éxito realizando sus unificaciones. No hay que buscar procedimientos, ni conservar las alternativas pendientes, ni contemplar la posibilidad de fallo.

```
procedure ejecucion_O_recomp(p:programa; R:resolvente;
            S:pila de backtracking: V:sustitucion; resultado:tipo_res);
var
    P_a: procedimiento asociado al predicado a;
    resultado:tipo_res;
    a : objetivo;
    c : clausula;
    V' : sustitución;
    C : camino de exito;
begin
   repeat
       a := R[1] /* Se toma el primer atomo del resolvente */
       buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
       repeat
          c := P_a[1]; /* Se toma la primera clausula del procedimiento */
          P_a := P_a - c;
          if paralelismoO(c) then
          begin
             meter(P_a[1],C);
             tareaO_recomp(p,R,S,V,C);
             P_a := P_a - P_a[1];
          end
          resultado := unificar(a,c,V');
       until (P_a = []) or resultado = exito;
       if resultado = exito then
       begin
          if P_a <> [ ] then salvar(P_a,R,V,S);
          V := componer(V,V');
          reemplazar(R,a,c); /* En R, reemplazar a por el cuerpo de c */
          meter(c,C); /* Se anota en C la alternativa explorada */
          a := R[1] /* Se toma el primer atomo del resolvente */
          buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
       end
       else if not vacio(S) then begin
          backtracking(S,a,P_a,R,V,C);
          resultado := exito;
       end
       else resultado := fallo
    until (R = []) or resultado = fallo;
end
```

Figura 2.12: Algorítmo de explotación del paralelismo_O por recomputación

```
procedure recomputar(p:programa; R:Resolvente; C:camino de exito;
V:sustitucion);
var
   a: objetivo;
   c: clausula;
begin
    while not vacio(C) do
    begin
       sacar(a,R);
       sacar(c,C);
       unificar(a,c,V)
       reemplazar(R,a,c); /* En R, reemplazar a por el cuerpo de c */
    end
end
```

Figura 2.13: Recomputación

Optimización por coincidencia

Si a un procesador se le asigna una tarea con un entorno que tiene alguna parte común con el entorno actual del procesador, sólo habrá que reconstruir la parte diferente. PDP aprovecha la coincidencia comparando el camino de éxito recibido con el que tenía el procesador y que correspondia a la última tarea asignada, evitando la recomputación de la parte común. Esta alternativa no reduce la información transmitida pero disminuye el tiempo empleado en la recomputación. La política de planificación tiene en cuenta ésta optimización dando prioridad al intercambio de trabajo entre procesadores que lo han intercambiado previamente. Si dos procesadores acaban de compartir un trabajo tendrán probablemente una parte común.

2.4 Combinación del paralelismo

Consideramos ahora programas que presentan ambos tipos de paralelismo. Cuando en una tarea_O aparece paralelismo_Y, el modelo de explotación de este último no requiere modificación ya que la tarea_O produce un entorno de trabajo identico al de la ejecución secuencial y pueden crearse tareas_Y que ejecuten los objetivos paralelos. Por el contrario, cuando en una tarea_Y aparece paralelismo_O, se presenta una nueva situación, ya que

la tarea padre de la tarea_Y espera una sola solución al objetivo cedido, para ir formando las combinaciones de soluciones a los objetivos de la llamada paralela mediante backtracking. Si se explota el paralelismo_O de la tarea_Y de forma que se obtengan simultaneamente las distintas soluciones al objetivo será necesario que o bien la tarea padre las almacene para formar el producto cruzado, o bien las tareas hijas las almacenen hasta que sean requeridas por la tarea padre. PDP evita almacenar las distintas soluciones y sincronizar la ejecución de las tareas creando una tarea_O para cada combinación de soluciones, de forma que no hay una tarea encargada de realizar el producto cruzado sino que se realiza de forma distribuida. El modelo de PDP para explotar paralelismo combinado cuando aparece en forma de *O_bajo_Y* se basa en que la recomputación permite a las tareas_Y explotar paralelismo_O creando tareas_O. De esta forma el el paralelismo_Y se explota con el esquema de explotación del paralelismo_O que reduce la interacción entre tareas (en la explotación del paralelismo_Y la tarea padre espera a que las tareas_Y creadas terminen y den una respuesta mientras que al explotar paralelismo_O se crean tareas_O autonomas) y por tanto mejora la eficiencia. Para realizar este esquema de formación del producto cruzado hay que tener en cuenta los siguientes requisitos:

- Las tareas_Y deben disponer de la información necesaria para crear una tarea_O: el objetivo inicial y el camino de éxito.
- Puesto que no hay ninguna tarea encargada de realizar el producto cruzado, es necesario establecer un criterio que permita a cada nueva tarea conocer la parte del árbol de búsqueda que le corresponde explorar.

2.4.1 Producto cruzado de alternativas en una Llamada Paralela

La Figura 2.14 representa el árbol de tareas correspondiente a la ejecución de la llamada paralela del siguiente programa:

```
:- p & q.
p :- p1.     q :- q1.
p :- p2.     q :- q2.
p :- p3.
```

Al buscar una solución a la llamada paralela obtenemos en primer lugar la que corresponde a las alternativas $p1$ y $q1$ de p y q respectivamente. Otras soluciones asociadas a los distintos elementos del producto cruzado son: $(p1, q2)$, $(p1, q3)$, $(p2, q1)$, etc. Al ejecutar el objetivo paralelo p, la tarea_Y T2 encuentra paralelismo_O, por lo que mientras se ocupa de explorar la primera de la alternativas, p_1, encarga la tarea de explorar una nueva solución a la tarea T4, que no considera la alternativa p_1 de p.

Para describir el criterio de reparto de los elementos del producto cruzado se introduce el concepto de *objetivo de procedencia* de una tarea T. LLamaremos al objetivo que presenta paralelismo_O y origina la creación de tareas_O, objetivo de procedencia de dichas tareas. En la Figura 2.14, p es el objetivo de procedencia de $T6$ y $T7$ y q es el objetivo de procedencia de $T8$ y $T9$. Los elementos del producto cruzado que se asignan a la nueva tarea se forman tomando para cada objetivo paralelo la cláusula alternativa dada por la siguiente *regla de combinación*:

- Si el objetivo está a la izquierda del de procedencia, la rama a explorar se *fija* a la misma explorada por la última tarea_O antepasada.
- Si el objetivo es el de procedencia, la rama a explorar es la siguiente a la explorada por la tarea_Y padre.
- Si el objetivo está a la derecha del de procedencia, la rama a explorar es la que lleva a la primera solución.

De esta forma se consigue que cada tarea fije las alternativas a tomar para los objetivos a la izquierda del de procedencia y las combine con las distintas alternativas de los restantes objetivos de la llamada paralela.

2.5 Modelo de ejecución de PDP

PDP explota paralelismo_Y puro, paralelismo_O puro y la combinación de ambos, produciendo *tareas_O*, que exploran las ramas del árbol de búsqueda desde la raíz y *tareas_Y*, que ejecutan objetivos paralelos. El modelo de ejecución se PDP puede sintetizarse de la siguiente forma:

- La ejecución de un programa comienza como una tarea_O que registra el camino de éxito seguido.

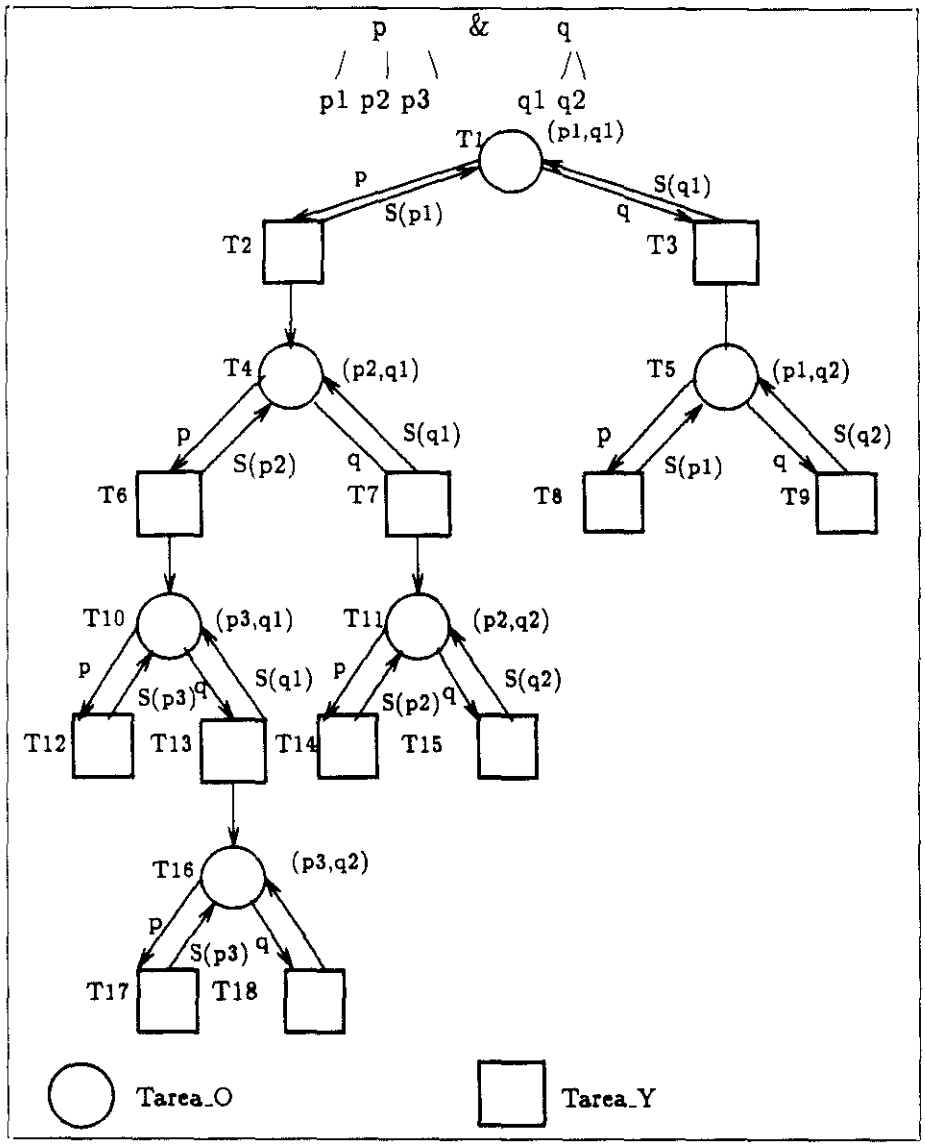


Figura 2.14: Explotación conjunta del paralelismo

- Cuando aparece paralelismo_O se crean tareas_O que recomputan el objetivo inicial siguiendo el camino de éxito de la tarea padre.

- Cuando aparece paralelismo_Y se crean tareas_Y que ejecutan los objetivos paralelos y de las que la tarea padre recoge el resultado.

- Si una tarea_Y encuentra paralelismo_O, se crean nuevas tareas_O encargadas de las cláusulas paralelas a las que transfiere el camino de éxito que lleva a la llamada paralela. Esta es la razón por la que las tareas_Y reciben el camino de éxito. De esta forma, **el paralelismo_Y se explota con el esquema de explotación del paralelismo_O**, lo que reduce el intercambio de información.

- Las tareas_O creadas por tareas_Y, al terminar la recomputación del camino de éxito recibido, deciden mediante la *regla de combinación* la parte del árbol de búsqueda que les corresponde explorar.

En el modelo de ejecución se introduce una nueva estructura, el *entorno de producto cruzado*, que indica la parte del camino de éxito correspondiente a cada uno de los objetivos de una llamada paralela.

2.6 Tareas de PDP

El esquema de explotación del paralelismo_O_bajo_Y de PDP lleva a distinguir distintos tipos de tareas_O y tareas_Y, dependiendo del tipo de tarea de la que surgen y de la posición del objetivo de procedencia. Los tipos de tareas son:

- **Tarea_O primaria:**  Se origina para la explotación del paralelismo_O en una tarea_O. Cuando la recomputación del camino de éxito recibido termina, la ejecución continúa normalmente.

- **Tarea_O secundaria:**  Se origina para la explotación del paralelismo_O en una tarea_Y. Cuando la recomputación del camino de éxito, que lleva a la llamada paralela de la que procede, termina, se crea una nueva combinación de soluciones de acuerdo con el objetivo de procedencia.

- **Tarea_Y primaria:**  Se origina para la explotación del paralelismo_Y en una tarea_Y, una tarea_O_primaria o una tarea_O_secundaria si no corresponde a un objetivo a la izquierda del de procedencia. Las tareas_Y primarias explotan el paralelismo_O que encuentran durante la computación.

- **Tarea_Y secundaria:**  Se origina para la explotación del paralelismo_Y en una tarea_O_secundaria correspondiente a un objetivo de la izquierda del objetivo de procedencia. De acuerdo con la regla de combinación este tipo de tarea ignora el paralelismo_O que aparece durante la ejecución.

Según esta última clasificación de tareas, el árbol de tareas correspondiente a la ejecución de la llamada paralela de la Figura 2.14 pasa a ser el de la Figura 2.15. La tarea_O $T1$ en la que se inicia la ejecución es la única *primaria* en este ejemplo, ya que las restantes tareas_O se han originado a partir de tareas_Y. $T1$ crea las tareas_Y *primarias* $T2$ y $T3$ a las que asigna los objetivos p y q respectivamente. Cuando $T2$ y $T3$ encuentran paralelismo_O, lo explotan creando las tareas_O *secundarias* $T4$ y $T5$. Aplicando la *regla de combinación*, $T4$ explora la siguiente alternativa pendiente de p, es decir, $p2$, combinandola con todas las soluciones de q. Para la ejecución de los objetivos $T4$ crea tareas_Y *primarias*, ya que estos objetivos no estan a la izquierda del objetivo de procedencia. Por su parte $T3$ *fija* la alternativa a explorar para el objetivo p, que está a la izquierda del de procedencia, a $p1$ que es la alternativa explorada por la última tarea_O antepasada $T1$, y para q que es el objetivo de procedencia toma la siguiente alternativa pendiente $q2$. $T5$ crea tareas_Y *secundarias* (que no explotan paralelismo_O), ya que p está a la izquierda del objetivo de procedencia y a q no le quedan alternativas pendientes de explorar. El resto de la ejecución se desarrolla análogamente.

2.7 Algoritmo de creación de una tarea_Y

El algoritmo de creación de una *tarea_Y* aparece en la Figura 2.16. La entrada a la tarea está formada por el objetivo Q y la vinculación de sus variables V, el camino de éxito C y el *entorno de producto cruzado* CPE correspondiente a la llamada paralela. Se reciben también el programa p y el tipo de tarea_Y que debe crearse *tipotarea*. En primer lugar se inicializan el resolvente R al objetivo a ejecutar Q y la pila de backtracking S a vacio. La

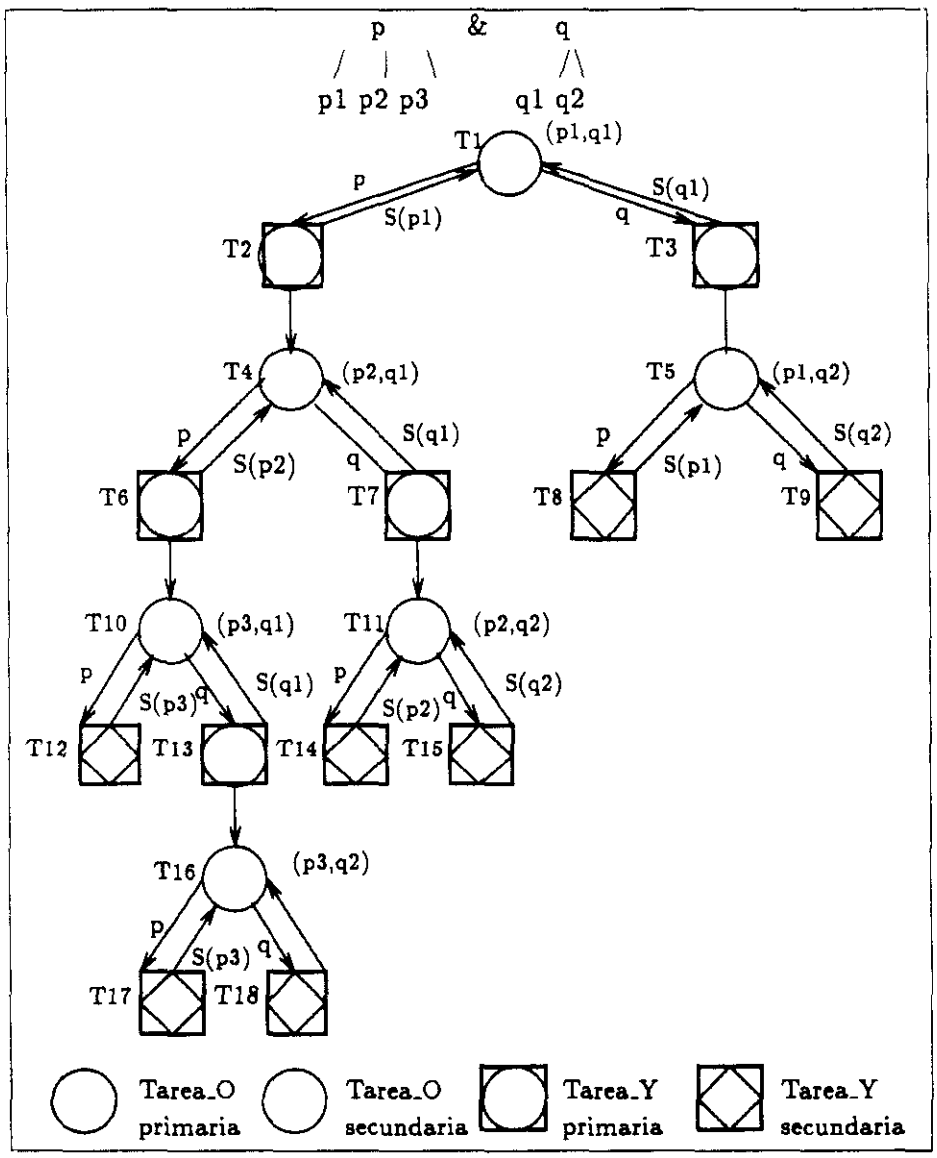


Figura 2.15: Ejemplo de explotación del paralelismo en PDP

```
type tipo_tarea = (O_primaria, O_secundaria, Y_primaria, Y_secundaria);
process tarea_Y(p:programa; Q:objetivo; V:sustitucion;C:camino de exito;
                EPC: entorno_producto_cruzado; tipotarea: tipo_tarea);
var
    R: resolvente;
    S: pila de backtracking;
    resultado: tipo_res;
begin
    R := [Q];
    S := ∅;
    C := C[EPC[Q]];
    recomputacion(p, R, S, V, C, EPC);
    if R <> ∅ then begin
        /* se actualiza el EPC */
        objetivo_procedencia(EPC) := Q;
        ejecucion(p,R,S,V,C,EPC,tipotarea,resultado);
    end
    else
        resultado := exito
    if resultado = exito then
        enviar_exito(tarea_padre, restriccion(V,Q), alternativa);
    else
        enviar_fallo(tarea_padre);
end
```

Figura 2.16: Algorítmo de creación de una tarea Y en PDP

nueva tarea_Y toma del *camino de éxito* recibido C la parte correspondiente al objetivo Q, que comienza en el punto indicado por el *entorno de producto cruzado* EPC. El camino de éxito correspondiente al objetivo puede estar vacio, incompleto o completo dependiendo del tipo de tarea y de la posición que ocupa el objetivo en la llamamada paralela. Durante la recomputación de este camino de éxito dependiendo del tipo de tarea_Y de que se trate se explota o no el paralelismo_O que aparece. Si al terminar la recomputación del camino de éxito correspondiente al objetivo, el *resolvente* R no está vacio, se realiza la ejecución pendiente. La respuesta obtenida al terminar la ejecución se envía a la tarea padre.

2.8 Algoritmo de creación de una tarea_O

```
proccess tareaO(p:programa;Q:objetivo:C:camino de exito;
                EPC: entorno_producto_cruzado; tipotarea: tipo_tarea);
var
    S : pila de backtracking;
    V : sustitucion;
    LP: resolvente;
    resultado:tipo_res;
begin
    S := ∅
    R := [Q]
    recomputar(p,R,C,V,C,EPC);
    if tipotarea = O_secundaria then begin
        LP := llamada_paralela(R);
        nueva_combinacion(p,LP,S,V,C,EPC,resultado);
    end
    ejecucion(p,R,S,V,C,EPC, tipotarea, resultado);
    if resultado = exito then begin
        V := restricción(V,Q);
        visualizar(V);
    end
end
```

Figura 2.17: Algoritmo de creación de una tarea_O

El algoritmo de creación de una *tarea_O* aparece en la Figura 2.17. El objetivo a resolver se *recomputa* siguiendo el camino de éxito de la tarea padre. Al terminar la ejecución, si se trata de una tarea_O *primaria* la

ejecución continúa normalmente. Si se trata de una tarea_O *secundaria* la recomputación ha llegado hasta una llamada paralela para la que se forma una nueva combinación de alternativas como aparece en la Figura 2.18. Los objetivos a la izquierda del de procedencia se ejecutan mediante tareas_Y *secundarias*, mientras los restantes se ejecutan por medio de tareas_Y *primarias*.

2.9 Algoritmo de ejecución de PDP

La Figura 2.19 presenta el algoritmo de ejecución de PDP. El *entorno* de ejecución está constituido por el *resolvente* R, la *pila de backtracking* S, la *sustitución de respuesta* V, el *camino de éxito* C y el *entorno de producto cruzado* EPC. La secuencia de pasos de resolución se repite hasta que el resolvente está vacio (ejecución con éxito) o se produce un fallo. Se distinguen dos tipos de pasos de resolución, dependiendo de la existencia de paralelismo_Y. Si hay paralelismo_Y se explota creando una tarea_Y paralela para la ejecución de cada uno de ellos. El tipo de la tarea_Y que se crea depende del tipo de la tarea padre y de la posición del objetivo asignado respecto del de procedencia. Cada una de estas tareas recibe el programa, el objetivo asignado y la vinculación de las variables del objetivo (*restriccion*(V, A_i)), el camino de éxito y el entorno de producto cruzado que les indica que parte del camino de éxito corresponde al objetivo. La computación de estas tareas_Y produce un resultado ($result_i$) de éxito o fallo. Si el resultado es de éxito, la tarea_Y proporciona la sustitución de respuesta computada correspondiente al objetivo (θ_i). Si el resultado de todas la tareas_Y es de éxito, para cada una de ellas aplica al resolvente pendiente la sustitución de respuesta computada (R := aplicar(θ_i,R)) y se compone dicha sustitución con la general($v := v \circ \theta$). Si todos los objetivos de la llamada paralela terminan con éxito, la función *completa* lo anota para tenerlo en cuenta en caso de backtracking. El otro tipo de paso de resolución se dá cuando no hay paralelismo_Y y se ejecuta un único objetivo. En este caso, se busca el procedimiento asociado al objetivo. Si las cláusulas de éste procedimiento presentan paralelismo_O, su explotación se realiza creando tareas_O secundarias si la tarea actual es una tarea_Y o una tarea_O secundaria, y tareas_O primarias en otro caso. Las alternativas cedidas a las tareas_O creadas dejan de estar pendientes ($P_a := P_a - P_a[1]$). Si se consigue la unificación del objetivo con alguna de las cláusulas del procedimiento, se

```
procedure nueva_combinacion(p:programa; R: Resolvente; S:pila de bactracking;
V:sustitucion; C: camino de exito; EPC: entorno_producto_cruzado; resultado:tipo_res);
var
    r,i : integer;
    θ: array[1..NMAX] of sustitucion;
    result: array[1..NMAX] of tipo_res;
    OP : integer; /* objetivo de procedencia */
begin
    [A_1, ..., A_r] := R; r := size([A_1, ..., A_r]);
    OP := objetivo_procedencia(EPC);
    parbegin
    /* se crean tareas_Y_secundarias para obj. a la izq. del de procedencia */
       tarea_Y(p,A_1,restriccion(V,(A_1)),C,EPC,secundaria, result_1, θ_1);
                              .
                              .
                              .
       tarea_Y(p,A_{OP-1},restriccion(V,(A_{OP-1})),C,EPC,secundaria, result_{OP-1}, θ_{OP-1});
    /* se crean tareas_Y_primarias para los restantes objetivos */
       tarea_Y(p,A_OP,restriccion(V,(A_OP)),C,EPC,primaria, result_OP, θ_OP);
                              .
                              .
                              .
       tarea_Y(p,A_r,restriccion(V,(A_r)),C,EPC,primaria, result_r, θ_r);
    parend
    i := 1;
    while (i <= r) and resultado = exito do begin
       resultado := result_i;
       if resultado = exito then begin
          R := aplicar(θ_i,R);
          V := V ∘ θ_i;
       end
    end
    if resultado = exito then begin
       completa(L); salvar(L,S);
    end
end
```

Figura 2.18: Algorítmo de formación de una nueva Combinación de alternativas.

```
procedure ejecucion(p:programa; R:resolvente; S:pila de backtracking;
                    V: sustitucion; C:camino de éxito; EPC: entorno_producto_cruzado;
                    tipotarea: tipos_tarea; resultado:tipo_res);
var
    a : objetivo;      P_a: procedimiento asociado al predicado a;
    c : clausula;      V' : sustitución;      r, i : integer;
    θ: array[1..NMAX] of sustitucion; result: array[1..NMAX] of tipo_res;
begin
   repeat
       [A_1, ..., A_r] := conjunto_independiente(R); r := size([A_1, ..., A_r]);
       if r > 1 then begin
          parbegin /* PARALELISMO Y */
             if tipotarea = tarea_O_primaria or tipotarea = tarea_Y_primaria or
             tipotarea = tarea_O_secundaria and objetivo_proce(EPC) >= 1 then
                tarea_Y(p,A_1,restriccion(V,(A_1)),C,EPC,primaria, result_1, θ_1);
             else
                tarea_Y(p,A_1,restriccion(V,(A_1)),C,EPC,secundaria, result_1, θ_1);
                                        .
                                        .
                                        .
          parend
          i := 1;
          while (i <= r) and resultado = exito do begin
             resultado := result_i;
             if resultado = exito then begin
                R := aplicar(θ_i,R);
                V := V o θ_i;
             end
          end
          if resultado = exito then completa(L);
       end
       else begin /* r = 1 */
          a := R[1] /* Se toma el primer atomo del resolvente
          buscar_procedimiento(a,p,P_a); /* procedimiento asociado al atomo a */
          repeat
             c := P_a[1]; /* Se toma la primera clausula del procedimiento */
             P_a := P_a - c;
             if paralelismoO(c) then begin
                if tipotarea = tarea_Y or tipotarea = tarea_O_secundaria then
                   tarea_O(p,R,V,C,EPC,secundaria);
                else if tipotarea = tarea_O then
                   tarea_O(p,R,V,C,EPC,primaria);
                P_a := P_a - P_a[1]; end
             resultado := unificar(a,c,V');
          until (P_a = []) or resultado = exito;
          if resultado = exito then begin
             if P_a <> [ ] then salvar(P_a,R,V,S);
             V := componer(V,V');
             reemplazar(R,a,c); /* En R, reemplasar a por el cuerpo de c */
             meter(c,C); end /* Se anota en C la alternativa explorada */
       end
       if resultado = fallo then backtracking(p,S,R,V,C,resultado);
   until (R = []) or resultado = fallo;
end
```

Figura 2.19: Algorítmo de ejecución de PDP

almacena el estado del entorno en la pila de backtracking, se compone la sustitución obtenida con la general y se reemplaza en el resolvente el átomo unificado por el cuerpo de la cláusula con la que se ha unificado. Si al dar un paso de resolución produce un resultado de fallo se realiza *backtracking* recuperando un estado anterior del entorno de la pila de backtracking.

2.10 Anotación del paralelismo

PDP sólo explota el paralelismo indicado por anotaciones en el programa. Estas anotaciones pueden realizarse bien por el usuario o bien por un precompilador. Por ejemplo en el caso del paralelismo_Y puede disponerse de un precompilador que realice un análisis de independencia de variables. El paralelismo_Y se anota mediante el operador & entre los objetivos de la llamada paralela:

$$:-(p(X,Y)\&q(X,Z)),...$$

El paralelismo_O se anota mediante el símbolo * delante de las cláusulas de un procedimiento que se desea ejecutar en paralelo. El siguiente fragmento de código anotado:

```
*p :- p1
*p :- p2
 p :- p3
```

indica que la primera cláusula se ejecutará en paralelo con la segunda y la segunda con la tercera. A partir de estas anotaciones el compilador produce un código intermedio con el paralelismo expresado mediante las nuevas instrucciones que se introducen.

También es posible indicar al sistema que ejecute secuencialmente una parte del programa, de forma que aunque se llame a un procedimiento anotado con paralelismo, se ejecute secuencialmente. La forma de hacerlo es introducir un objetivo previo *sec* a partir del cual deja de explotarse paralelismo hasta que aparece un objetivo *nosec*. Así para el siguiente programa:

```
:- seq, (p&q), nosec.

*p :- p1.
*p :- p2.
```

```
p :- p3.

*q :- q1.
 q :- q2.
```

no se explota el paralelismo_Y de la llamda paralelela ($p\&q$) ni el paralelismo_O de los procedimientos p y q.

3

Procesadores básicos de PDP

3.1 Introducción

En este capítulo se describe la implementación del modelo de ejecución de los procesadores básicos de PDP. Se ha tenido como objetivo de diseño mantener las técnicas de la WAM en los segmentos secuenciales de programa conservando así las optimizaciones ya conseguidas en Prolog secuencial. De la misma forma se ha tratado de mantener las técnicas de explotación de cada tipo de paralelismo con las menores modificaciones. Presentamos en primer lugar la relación de los nuevos aspectos que presenta el modelo de ejecución respecto al secuencial y que justifican la arquitectura de la máquina abstracta resultante. A continuación se describe la arquitectura de un procesador básico, es decir, las estructuras de datos e instrucciones que han aparecido como consecuencia de la explotación del paralelismo. Se describen también los procedimientos de intercambio de trabajo entre procesadores básicos.

3.2 Procesador básico: Extensión de la WAM

Las diferencias entre el algoritmo de la ejecución secuencial presentado en el Capítulo 1 con el de ejecución en PDP presentado en el Capítulo 2 indican que extensiones a la WAM requiere la implementación de PDP:

- **Registro del camino de éxito.**
  El algoritmo de ejecución de PDP distingue dos tipos de pasos de

resolución: el que computa en paralelo un conjunto de objetivos independientes del resolvente y el que computa un único objetivo. Si comparamos un paso de resolución de un único objetivo con la resolución secuencial, vemos que las diferencias consisten por una parte en la consideración del paralelismo_O al buscar la cláusula que encaja con el objetivo a resolver, y por otra parte, en el registro en el camino de éxito de la cláusula seleccionada. Los procesadores básicos tienen que anotar el camino seguido en el árbol de búsqueda, actualizandolo cada vez que se toma una nueva alternativa o que se produce backtracking.

- **Control de las llamadas paralelas.**
  El otro tipo de paso de resolución, que considera un conjunto de objetivos independientes no existía en la ejecución secuencial, y por tanto requiere extender la WAM para su implementación. La ejecución de la llamada paralela se realiza creando tareas_Y del tipo que corresponda según el tipo de la tarea padre y la posición del objetivo de procedencia, para cada objetivo independiente. La ejecución de cada uno de los objetivos de una llamada paralela se sincroniza para continuar la ejecución solo cuando todos ellos han sido computados con éxito.

- **Backtracking**
  Las acciones a realizar cuando se produce un fallo dependen del tipo de tarea que está realizando el procesador básico. El mecanismo de backtracking se extiende para considerar las distintas situaciones.

- **Recomputación**
  Al crear una tarea_O se recomputa el *camino de éxito* recibido de la tarea padre. Por tanto los porcesadores básicos tienen un modo de funcionamiento especial, en el que no seleccionan la cláusula con la que resolver el objetivo de acuerdo con la estrategia de búsqueda de Prolog, sino que toman la cláusula indicada en el camino de éxito.

- **Consideración de distintos tipos de funcionamiento.**
  A un procesador básico se le puede asignar cualquiera de los tipos de tareas descritos en el Capítulo 2. Por tanto el procesador debe distinguir entre distintos modos de funcionamiento según la tarea asignada en cada momento.

- **Formación de producto cruzado de las soluciones de los objetivos de una llamada paralela.**
  En la tareas_O *secundarias*, es decir, en las que han sido creadas para la explotación del paralelismo_O en una tarea_Y, la recomputación del camino de éxito termina en una llamada paralela para la que se requiere una nueva combinación de soluciones. Por tanto se introduce un nuevo procedimiento que crea esta combinación aplicando la *regla de combinación*.

- **Intercambio del trabajo pendiente.**
  La creación de las nuevas tareas consiste en la asignación de trabajos pendientes a procesadores básicos. Por tanto se necesitan procedimientos que seleccionen y envien la información que corresponde en cada caso.

A continuación se describe la implementación de cada uno de estos puntos.

3.2.1 Registro del camino de éxito

Para registrar el *camino de éxito* introducimos una nueva pila en cada procesador (pila de camino de éxito) y un registro asociado RE que apunta a su cima, denominado "registro de éxito". Además introducimos un nuevo campo en los puntos de elección con el valor del "registro de éxito", para la actualización automática de la pila de éxito durante el backtracking.

Cuando se crea un punto de elección se coloca en la pila de éxito la dirección del código que se va a ejecutar, se anota el registro de éxito en el punto de elección y se actualiza dicho registro. Cuando se produce backtracking se actualiza el punto de elección con la dirección de la siguiente alternativa a probar y la dirección de la cima de la pila de éxito con la alternativa que está siendo probada actualmente. Cuando se toma la última alternativa, el punto de elección desaparece, pero la dirección de la pila de éxito asociada permanece con la dirección de dicha alternativa. Cuando se cede trabajo a otro procesador se le envía el camino de éxito contenido en la pila de éxito, cambiando la última dirección, que corresponde a la alternativa que está siendo explorada, por la de la siguiente alternativa a probar para que el procesador padre y el que recibe el trabajo exploren distintas rama a partir de ese punto.

Así para el programa:

```
G:   :- p.

P1:  p :- q, r, s.
P2:  p :- t, v.

Q1:  q :- ...
Q2:  q :- ...
Q3:  q :- ...

R1:  r :- ...
R2:  r :- ...
```

Podemos encontrar el estado de la máquina como se representa en la Figura 3.1. El contenido de las pilas indica que se ha tomado la primera cláusula del procedimiento P, la última del procedimiento Q (por lo que ha desaparecido el punto de elección asociado) y la primera del procedimiento R. Si ahora se cede trabajo a otro procesador, se le envía el camino: P1,Q3,R2 y se actualiza el punto de elección 2. De esta forma si se produce fallo no se intenta la alternativa R2, que ha sido cedida, sino la R3.

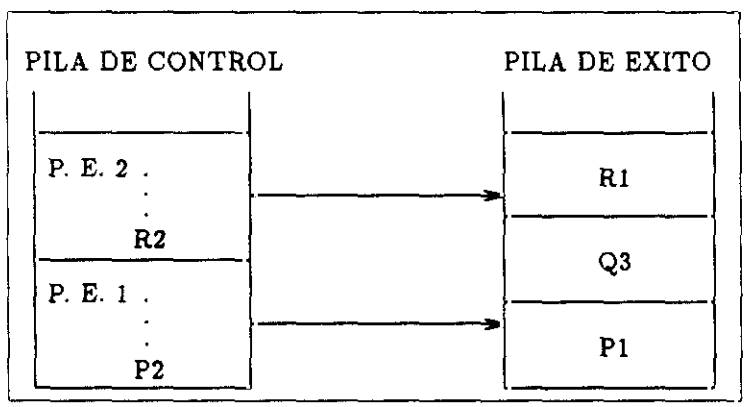


Figura 3.1: Recomputación

3.2.2 Control de una llamada paralela y Formación del producto cruzado de soluciones

Una vez seleccionados los objetivos independientes necesitamos extender la máquina para el tratamiento de la ejecución paralela de estos objetivos. Se introduce un conmutador de estado *conmu_Y* mediante el cual el procesador básico distingue entre un estado de ejecución secuencial (SEC) o de ejecución de una llamada paralela (PAR).

El registro del camino de éxito de los objetivos de una llamada necesita un tratamiento especial ya que el camino correspondiente a cada objetivo, que se recibe junto con la respuesta, puede llegar en un orden distinto al que ocupan los objetivos en una llamada paralela. Para acceder a los caminos de éxito de los objetivos de la llamada paralela, ordenandolos adecuadamente y seleccionandolos para ser asignados a una nueva tarea, se crea una estructura, el **Marcador de Producto Cruzado (MPC)**. La información registrada en un MPC es la siguiente:

- Por cada objetivo de la llamada paralela se registra el punto de la Pila de Exito donde comienza el camino de exito correspondientes
- Objetivo de procedencia
- Número de objetivos de la Llamada

El MPC permite la formación del producto cruzado de soluciones aplicando la regla de combinación. Los MPC's se almacenan en una pila llamada **Pila de Producto Cruzado**, cuya cima está señalada por el registro **Registro de Producto Cruzado**.

Se necesita también un mecanismo para controlar la ejecución compartida de los objetivos de la llamada y la reunión posterior de las vinculaciones que origina su ejecución. Este mecanismo de control de PDP es una extensión del propuesto por Hermenegildo [33]. La tarea padre crea una estructura en el Stack, la **Llamada paralela (LP)** en la que registra la evolución de los objetivos para los que crea una serie de tareas_Y.

La información registrada en una Llamada Paralela es la siguiente:

- Por cada objetivo de la llamada paralela se registra:
  - procesador al que se asigna la tarea_Y correspondiente

  - estado de la computación:
    * sin ejecutar
    * en ejecución
    * ejecutado dejando alternativas pendientes
    * ejecutado y sin alternativas

- Número de objetivos en ejecución
- Número de objetivos en espera de ser ejecutados
- Número de objetivos de la llamada paralela
- Valores de los registros de control del *Stack*
- Valor del registro de Producto Cruzado

El almacenamiento de los valores de los registros de control de la pila de Control y del registro de Producto Cruzado permite recuperar automáticamente estas pilas cuando se produce backtracking.

Una tarea_Y registra las variables libres del objetivo recibido y cuando acaba su ejecución devuelve a la tarea padre la sustitución de respuesta computada restringida a estas variables. Para una tarea_Y el objetivo recibido es el objetivo inicial y solo se identifica como tarea_Y mediante una nueva estructura de *Stack*, el **Marcador de Objetivo Remoto (MPR)**, en la que registra los identificativo de su procedencia:

- Identificativo de la tarea padre
- Llamada Paralela a la que pertenece el objetivo
- Orden del objetivo en la Llamada Paralela
- Valores de los registros de control de Stack(LGM,HReg,TRReg, RetReg,EnvReg), en el momento de la creación de la estructura

La tarea padre se ocupa también de la ejecución de los objetivos de la llamada que no han sido enviados a otros procesadores. Para identificarlos como objetivos pertenecientes a una Llamada Paralela, se marcan con una nueva estructura que se coloca en el *Stack*: el **Marcador de Objetivo Propio (MOP)**, que registra:

- Llamada Paralela a la que pertenece el objetivo
- Orden del objetivo en la Llamada Paralela
- Valores de los registros de control de Stack(LGM,HReg,TRReg, RetReg,EnvReg), en el momento de la creación de la estructura

La unión de todas las instanciaciones de las variables de los objetivos de la llamada paralela junto con la parte del resolvente pendiente de computar configuran el entorno de trabajo del procesador padre, que sigue su evolución según los mecanismos secuenciales. La reunificación se identifica para tener un tratamiento especial en el backtraking mediante una estructura: **Fin de Llamada Paralela (FLP)** y que contiene los valores de los registros de control de Stack(IGM,HReg,TRReg, RetReg,EnvReg), en el momento de la creación de la estructura.

3.2.3 Backtracking

Las estructuras introducidas en el apartado anterior permiten distinguir los distintos estados de procesamiento en que puede encontrarse una llamada paralela al producirse backtracking. Así si la llamada está *incompleta*, es decir, para alguno de los objetivos independientes no se obtenido ninguna solución, se hace backtracking a un punto anterior de la llamada. Si la llamada estaba *completa*, se busca algún objetivo con alternativas pendientes. Si este objetivo es remoto se comprueba que esté *relacionado* con el fallo producido según el mecanismo de BI introducido. Para ello se marcan las variables que forman parte de la sustitución de respuesta computada por tareas_Y. Para realizar esta marca, la variable instanciada por una tarea_Y se instancia a la primera posición libre de una tabla, **la tabla de dependencias**, y en dicha posición se coloca la instanciación junto con el identificativo del objetivo ejecutado en la tarea_Y. De esta forma, siempre que al consultar una variable se acude a la tabla de dependencias se sabe que el objetivo cuyo identificativo aparece en la posición accedida ha contribuido a la instanciación actual de dicha variable, lo que se señala en el **registro de dependencias** (vector de bits asociados a los objetivos de una llamada ejecutados en otros procesadores) activando el bit correspondiente al identificativo del objetivo. Cuando se necesita una nueva solución de un objetivo ejecutado por una tarea_Y para tratar un fallo, solo se realiza la petición

si el bit correspondiente a este objetivo en el registro de dependencias está activo.

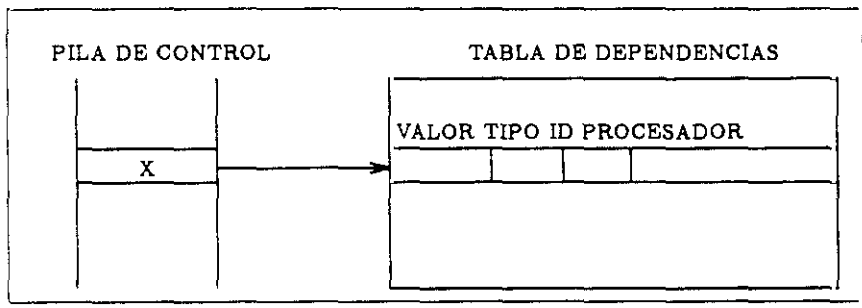


Figura 3.2: Implementación del BI

Esquemáticamente los puntos en que se modifica la máquina para la incorporación del Backtracking Inteligente son los siguientes:

- Cuando se recibe la sustitución de respuesta de una tarea_Y, las variables instanciadas por dicha respuesta se referencian a las entradas de la tabla de dependencias, en la que además se almacena el identificativo de la tarea_Y que las ha instanciado y el identificativo del objetivo.
- Se dispone de un registro adicional, registro de dependencias que es un vector de bits en el que cada bit representa uno de los subobjetivos ejecutados en otro procesador que han afectado a términos de la ejecución actual.
- Si al consultar una variable se acude a la tabla de dependencias, se señala en el registro de dependencias el bit correspondiente al objetivo de la posición de la tabla accedida.
- Cuando se produce un fallo y se llega a una llamada paralela durante el proceso de backtracking, se comprueba si el identificativo asociado a dicho objetivo está marcado en el registro de dependencias. Sólo si lo está se considera el backtracking de dicha llamada paralela.
- Durante la ejecución de la instrucción proceed, que finaliza la ejecución de un objetivo, se limpia el registro de dependencias.

3.2.4 Recomputación

Para realizar la recomputación se incorpora un conmutador de *modo* de funcionamiento cuyo estado puede ser DIRIGIDO, si el procesador sigue un camino de éxito recibido, LIBRE si está construyendo su propio camino. Se introduce también el *registro de recomputación* RR utilizado para recorrer el camino de éxito durante la recomputación.

3.2.5 Consideración de los distintos tipos de funcionamiento

Un procesador básico puede funcionar en distintos modos según se encuentre registrando o recomputando un camino de éxito dado, (*libre, dirigido*) o ejecutando código secuencial o una llamada paralela (*secuencial, paralelo*) y estos modos de funcionamientos pueden aparecer combinados. Cuando aparece paralelismo_O_bajo_Y, las tareas_Y encargadas de la ejecución de los objetivos de una llamada paralela encuentra paralelismo_O que explotan creando una tarea_O secundaria. Esta tarea recomputa el camino de exito recibido, funcionando en modo *dirigido* y *secuencial*. Si aparecen llamadas paralelas durante esta recomputación la máquina pasa a estar en estado *dirigido* e *Y_paralelo*, en el que explota el paralelismo_Y mediante tareas_Y primarios. Al concluir cada una de estas llamadas la máquina vuelve a estado *dirigido* y *secuencial*. La recomputación del camino de exito lleva a la llamada paralela a la que pertenece el objetivo cuyo paralelismo_O originó la tarea_O actual y para la que la tarea debe explorar una nueva combinación de soluciones. En este punto pueden darse diferentes situaciones ya que no se recibe el camino de exito de todos los objetivos de la llamada: no se recibe el camino de exito de los objetivos a la izquierda del objetivo de procedencia. Por esto, si al completar la recomputación del camino de éxito no pasamos a estado *libre*, si no ha un nuevo estado *combinación* en el que la máquina crea la nueva combinación de alternativas. Tenemos entonces los siguientes modos de funcionamiento de la máquina:

- **Libre y secuencial**
  Funciona como la WAM pero anotando el camino de éxito

- **Dirigido y secuencial**
  La tarea toma en cada procedimiento la cláusula indicada en el camino de éxito y no crea puntos de elección.

- **Libre y en modo de explotación del paralelismo_Y**
  La tarea comparte con otras tareas la ejecución de los objetivos de la llamada paralela, recibiendo de estas tareas la respuesta a la ejecución de los objetivos y el camino de éxito seguido para alcanzarla.

- **Dirigido y en modo de explotación del paralelismo_Y**
  La tarea comparte con otras tareas la ejecución de los objetivos de la llamada paralela, y les indica que camino de éxito deben seguir para obtener la solución correspondiente.

3.3 Arquitectura de un procesador básico

La arquitectura de un procesador básico está constituida por las estructuras de datos de que dispone junto con el conjunto de instrucciones que determinan su funcionamiento.

3.3.1 Estructuras de datos de un procesador básico

Las nuevas estructuras de datos que aparecen en la arquitectura como consecuencia de la explotación del paralelismo están relacionadas con las diferencias de PDP con el modelo de ejecución secuencial. Se introduce un registro que almacena el tipo de tarea. Las estructuras incorporadas para registrar el camino de éxito son:

- La "pila de éxito" y el "registro de éxito"

- Un nuevo campo en los puntos de elección con el contenido del registro de éxito al crearse el punto de elección.

Para realizar la recomputación se incorpora el conmutador de *modo* DIRIGIDO\LIBRE.

Para el control de las llamadas paralelas se han introducido las siguientes estructuras:

- Marcador de producto cruzado

- Llamada Paralela, para el reparto y reunificación de los objetivos de la llamada

- Marcador de objetivo remoto

- Marcador de objetivo Propio
- Fin de Llamada Paralela
- Conmutador *conmu_pary* que puede estar en modo secuencial (SEC), Y_paralelo (PAR)

Introducimos también las estructuras necesarias para el control del trabajo pendiente: la tabla de alternativas y su registro asociado y la pila de objetivos donde se almacenan los objetivos pendientes de ejecución. Reuniendo todos estos elementos tenemos el esquema de la arquitectura de la Figura 3.3.

3.4 Instrucciones de un procesador básico

PDP añade al repertorio de instrucciones de la WAM instrucciones para expresar el paralelismo_Y, bien de forma incondicional o bajo condiciones de indepedencia, para expresar paralelismo_O, e instrucciones para expresar paralelismo_Y con paralelismo_O en su explotación. Los tipos de datos que manejan estas instrucciones son básicamente direcciones (*dir*) y el tipo básico de los datos en PDP (*dato*), formado por un *valor* y un *tipo* (Figura 3.4) con el que se construyen las estructuras de PDP. El campo *valor* puede almacenar un dato, en forma de referencia a la tabla de símbolos, una variable en forma de referencia a sí misma, o una referencia a otro dato. El campo *tipo* indica si se trata de una referencia o un dato y en este caso si es una variable, constante o una estructura.

3.4.1 Instrucciones para la explotación del paralelismo_Y

El análisis realizado en tiempo de compilación permite saber a partir de comprobaciones sencillas en tiempo de ejecución que objetivos cumplen la condición de independencia y pueden ser ejecutados en paralelo. Estos objetivos y las comprobaciones de las que depende su ejecución paralela se anotan en el código de entrada a PDP mediente las nuevas instrucciones:

par_exec
check_ground
check_independent
alloc_p
callLocal

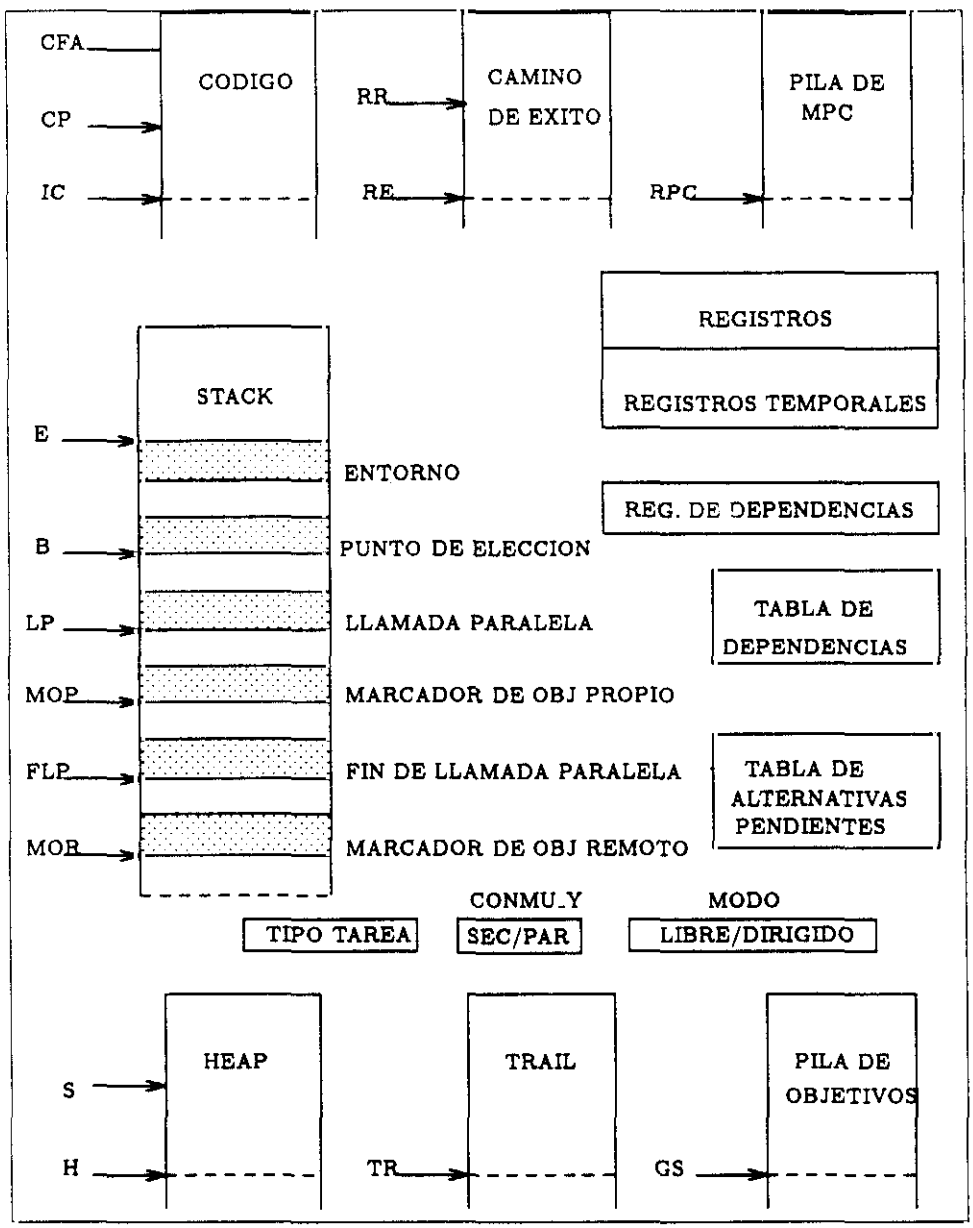


Figura 3.3: Estructuras de datos de un procesador básico

Figura 3.4: Dato básico utilizado en la implementación de PDP

pop_goal
wait

Estas instrucciones se basan en las propuestas por Hermenegildo [33] en su máquina pero utilizando un único código para la ejecución paralela y secuencial de las llamadas paralelas. Este código funciona en una u otra forma según el estado de un conmutador SEC/PAR. La instrucción *par_exec* cambia el estado de la máquina a ejecución paralela. Si el paralelismo detectado es incondicional a continuación se realiza la llamada paralela. Si es condicional, a esta instrucción le siguen las que realizan el analisis de cerrazón e independencia en tiempo de ejecución (*check_ground* y/o *check_independent*). Si alguna de estas comprobaciones falla, la máquina vuelve a modo secuencial. El código paralelo que sigue comienza con una instrucción *alloc_p* que anota la información necesaria para el control de la llamada paralela (número de subobjetivos,...) y que no tiene efecto en modo secuencial. Les siguen las correspondientes a los subobjetivos: las instrucciones *put* que preparan la llamada, y la instrucción *call* que la realiza. Esta funciona como en la WAM si el modo es secuencial, y si es paralelo pone el subobjetivo a disposición de los procesadores desocupados del sistema. La instrucción *callLocal* es una optimización de la instrucción *call* para el caso del último objetivo, que es ejecutado en el propio procesador. Después de las instrucciones correspondientes a cada subobjetivo de la llamada paralela encontramos la instrucción *pop_goal* que toma un objetivo pendiente del propio procesador. Y por último encontramos la instrucción *wait* que espera el final de la ejecución de los objetivos tomados por otros procesadores. Así el código correspondiente a la llamada paralela:

$$:-f(X,Y)\&g(X,Z),...$$

sería:

```
par_exec
check_ground X
check_independent Y, Z
alloc_p 2, n

put_var X
put_val Y
call f,0

put_val X
put_val Z
call g,1

pop_goal
wait
proceed
```

Describimos a continuación el funcionamiento de cada una de estas instrucciones.

par_exec

Esta instrucción cambia el estado de la máquina a estado de explotación del paralelismo_Y (conmu_Y = PAR), lo que determina la forma de funcionamiento de las instrucciones de explotación del paralelismo_Y y de la instrucción *call*.

```
procedure par_exec
begin
    conmu_Y := PAR
end
```

Figura 3.5: Instrucción par_exec

check_ground

Esta instrucción comprueba la *cerrazón* de una variable si la máquina está en estado de explotación de paralelismo_Y. Si no se cumple dicha condición la máquina pasa a estado de ejecución secuencial. Si la máquina se encontraba en estado secuencial la instrucción no hace nada.

```
procedure check_ground (x:dato);
var v:dato;
begin
    if conmu_Y = PAR then
    begin
        v := desref(x);
      if not ground(v) then
          conmu_Y := SEC;
    end
end
```

Figura 3.6: Instrucción check_ground

check_independent

Esta instrucción funciona como la anterior, pero para la comprobación de la condición de independencia entre dos variables.

```
procedure check_independent (x,y:dato);
var v,u:dato;
begin
    if conmu_Y = PAR then
    begin
        v := desref(x); u := desref(y);
      if not independent(v,u) then
          conmu_Y := SEC;
    end
end
```

Figura 3.7: Instrucción check_independent

alloc_p

Esta instrucción reserva espacio en memoria para la estructura utilizada para controlar la ejecución paralela: la *llamada_paralela* LP. Salva el valor de los registros que se almacenan en la estructura e inicializa los restantes campos.

```
procedure alloc_p(n_objetivos: integer; n_var: integer);
begin
    LP := cima_pila(); /* se crea la estructura */
    for i := 1 to n_objetivos do
        LP[i].estado := pendiente;
    num_obj_pend(LP) := n_objetivos;
    num_obj_espera(LP) := 0;
    num_obj(LP) := n_objetivos;
    salvar_registros(LP);
    IC := dir;
    /* se crea el marcador de producto cruzado */
    MPC[RPC][n_objetivos + 1] := n_objetivos;
    RPC := RPC + n_objetivos + 2;
end
```

Figura 3.8: Instrucción alloc_p

callLocal

Esta instrucción realiza la llamada al último objetivo de una llamada paralela, que ocupa la posición *slot*. Este objetivo no se pone a disposición de los procesadores desocupados del sistema y es ejecutado en primer lugar por el procesador. La instrucción crea un *marcado de objetivo propio* (MOP) en el *stack* y actualiza la llamada paralela (LP).

pop_goal

Esta instrucción toma de la pila de objetivos pendientes de ejecución el objetivo que ocupa la posición *slot* dentro de la llamada paralela, y después actua como call_local. El control de programa vuelve a esta instrucción después de la ejecución del objetivo , repitiendose el proceso hasta que no quedan objetivos pendientes de la llamada paralela que se está tratando.

wait

Esta instrucción se utiliza para esperar las respuestas a los objetivos de una llamada paralela que estan siendo ejecutados en otros procesadores. Cuando se reciben todas las respuestas, si estas son de éxito, se crea un *Marcador de Fin de LLamada Paralela (FLP)* en el *Stack*. La máquina pasa a estado

```
procedure call_local(dir: direccion; slot :integer; n_var:integer);
begin
    MOP := cima_pila(); /* se crea el marcador */
    llamada_paralela(MOP) := LP;
    objetivo(MOP) := slot;
    salvar_registros(MOP);
    LP[slot].procesador := id_procesador;
    LP[slot].estado := en_ejecucion;
    num_obj_pend(LP) := num_obj_pend(LP) - 1;
    IC := dir;
    /* se actualiza el marcador de producto cruzado */
    MPC[RPC][slot] := RE;
end
```

Figura 3.9: Instrucción call_local

secuencial (SEC).

3.4.2 Instrucciones modificadas para la explotación del Paralelismo_Y

Para implementar la explotación del paralelismo_Y han sido modificadas las instrucciones de la WAM:

call
proceed

call

Esta instrucción tiene dos modos de funcionamiento dependiendo del estado de la máquina. Si se está ejecutando una llamada paralela (conmu_Y = *PAR*) coloca el objetivo a tratar en la pila de objetivos para que pueda ser cedido a otros procesadores. En modo secuencial (conmu_Y = SEC) funciona como en la WAM, saltando al procedimiento indicado en la instrucción.

proceed

Esta instrucción realiza una distinción de casos según el valor del registro de retorno de llamadas *RetReg* que le indica si se trata del final de un objetivo

```
procedure pop_goal();
begin
   if hay_obj(pila_obj) then
   begin
      /* se toma el obetivo de la pila */
      sacar(pila_obj, LP);
      sacar(pila_obj, dir);
      sacar(pila_obj, n_objetivos);
      sacar(pila_obj, slot);
      for i := 1 to n_objetivos do
         sacar(pila_obj, registro[i]);
      MOP := cima_pila(); /* se crea el marcador */
      llamada_paralela(MOP) := LP;
      objetivo(MOP) := slot;
      salvar_registros(MOP);
      LP[slot].procesador := id_procesador;
      LP[slot].estado := en_ejecucion;
      num_obj_pend(LP) := num_obj_pend(LP) - 1;
      IC := dir;
      /* se actualiza el marcador de producto cruzado */
      /* con el valor del camino de exito */
      MPC[RPC][slot] := RE;
   end
end
```

Figura 3.10: Instrucción pop_goal

```
procedure wait();
begin
    FLP := cima_pila(); /* se crea el marcador */
    num_obj_pend(LP) := 0;
    num_obj_esp(LP) := 0;
    conmu_Y := SEC;
end
```

Figura 3.11: Instrucción wait

```
procedure call(dir: direccion; slot :integer; n_var:integer);
begin
   if conmu_Y = SEC then
    begin
       StaReg := n_var;
       IC := dir;
    end
    else
    begin
       meter_objetivo(pila_objetivos);
    end
end
```

Figura 3.12: Instrucción call

perteneciente a una llamada paralela y si se ha ejecutado en el procesador padre o en otro. Si se trataba de un objetivo perteneciente a otro procesador ($RetReg = MOR$) se envia a este procesador la vinculación que ha producido la ejecución del objetivo sobre las variables recibidas. Si se trataba de un objetivo propio perteneciente a una llamada paralela ($RetReg = MOP$) se actualiza el marcador de la llamada paralela. En otro caso se devuelve el control al punto en que se realizó la última llamada.

3.4.3 Instrucciones para la explotación del paralelismo_O

PDP permite señalar los puntos del programa en que se desea explotar el *paralelismo_O* de dos formas: mediante las instrucciones *_par* o mediante las instrucciones *_exec*:

try_par
retry_par
try_exec
retry_exec

Estas intrucciones sustituyen a *tryMeElse* y *retryMeElse*. La razón de la existencia de estos dos tipos de instrucciones es que cada uno de ellos dá lugar a un reparto de trabajo diferente entre el procesador padre y el procesador desocupado que recibe el trabajo: mientras que las instrucciones *_par* ceden una sola alternativa, las instrucciones *_exec* ceden todas las alternativas que

```
procedure proceed();
begin
   if RetReg = MOR then
    begin
       enviar_respuesta(MOR);
    end
    else if RetReg = MOP then
    begin
       LP := llamada_paralela(MOP);
       num_obj_esp(LP) := num_obj_esp(LP) - 1;
    end
    else
    begin
       IC := RetReg;
    end
end
```

Figura 3.13: Instrucción proceed

se encuentras a partir de la dirección que figura como último parametro de la instrucción. Por ejemplo, para el siguiente fragmento de programa:

```
*p :- q1.
*p :- q2.
 p :- q3.
 p :- q4.
```

el código compilado correspondiente, considerando la explotación del paralelismo_O sería:

```
p\_1    try_par 0 p\_2
        call q1
        proceed
p\_2    retry_par q2
        call q2
        proceed
p\_3    retryMeElse p\_4
        call q3
        proceed
p\_4    trustMeElseFail
        call q4
        proceed
```

Estas instrucciones indican al procesador que realiza la computación que la cláusula correspondiente a la instrucción *_par* y la siguiente pueden ser ejecutadas en paralelo. En el ejemplo, $q1$ y $q2$ se explorarían simultaneamente por estar el código correspondiente a p_1 precedido por *try_par*, al igual que $q2$ y $q3$ por estar el código correspondiente a p_2 precedido por *retry_par*. Sin embargo, $q3$ y $q4$ no se exploran en paralelo. De esta forma cuando se quiere considerar la ejecución paralela de un procedimiento no es necesario hacerlo para todas sus cláusulas, sino sólo para aquellas que interese.

Otra posibilidad de marcar el paralelismo es utilizar las instrucciones *try_exec* y *retry_exec*:

```
p\_1    try_exec 0 p\_2 p\_3
        call q1
        proceed
p\_2    retryMeElse p\_3
        call q2
        proceed
p\_3    retryMeElse p\_4
        call q3
        proceed
p\_4    trustMeElseFail
        call q4
        proceed
```

Estas instrucciones indican que el conjunto de cláusulas que se encuentran a continuación de la instrucción *_exec* y el conjunto que se encuentra a partir de la cláusula cuya dirección es el último parametro de la instrucción, pueden ser ejecutados en paralelo. Es decir, no se cede una sola alternativa, sino un conjunto de ellas. Por tanto, la distinción entre las intrucciones *_par* y las *_exec* permite optar entre distintas formas de reparto de alternativas según convenga en cada programa.

3.4.4 Instrucciones par

Son las instrucciones que sustituyen a *tryMeElse* y *retryMeElse* cuando se quiere explorar en paralelo la solución que corresponde a la cláusula alternativa siguiente a la actual.

Instrucción try_par

Esta instrucción funciona como *tryMeElse*, pero anotando en la tabla de trabajos pendientes la dirección del punto de elección que se crea y avisando del trabajo pendiente al controlador. Como todas las intrucciones *try_* su funcionamiento depende del modo de la máquina. En modo *LIBRE* anota el camino de éxito. En modo *DIRIGIDO* sigue el camino de éxito mediante el registro RR y no crea puntos de elección. Si el camino de éxito indica una cláusula distinta (*camino_exito*[RR] ¡¿ IC) se salta a ella. Finalmente se comprueba si se ha terminado la recomputación (RR = RE) en cuyo caso se pasa a modo LIBRE.

```
procedure try_par(n_var:integer);
begin
   if modo = LIBRE then
   begin
      PE := cima_stack();
      crear_punto_eleccion(PE);
      camino_exito[RE] := IC;
      RE := RE + 1;
      tabla_trabajos_pend[n_trabajos_pend].dir := PE;
      n_trabajos_pend := n_trabajos_pendientes + 1;
      enviar_aviso(controlador,trabajo_pendiente);
   end
   else /* modo DIRIGIDO */
   begin
      if camino_exito[RE] <> IC then
         IC := camino_exito[RE];
      else
      begin
         RR := RR + 1;
         if RR = RE then
            modo = LIBRE;
      end
   end
end
```

Figura 3.14: Instrucción try_par

Instrucción retry_par

En modo LIBRE funciona como *retryMeElse*, actualizando el punto de elección de la cima de la pila y el camino de éxito. En modo DIRIGIDO comprueba si se ha terminado la recomputación del camino de éxito para pasar al modo correspondiente.

```
procedure retry_par();
begin
    if mod = LIBRE then
    begin
        actualizar_punto_eleccion(IC);
        camino_exito[RE - 1] := IC;
        tabla_trabajos_pend[n_trabajos_pend].dir := PE;
        n_trabajos_pend := n_trabajos_pendientes + 1;
        enviar_aviso(controlador,trabajo_pendiente);
    end
    else /* modo DIRIGIDO */
    begin
        RR := RR + 1;
        if RR = RE then
            if conmu_Y = SEC then
                modo = LIBRE;
    end
end
```

Figura 3.15: Instrucción retry_par

3.4.5 Instrucciones exec

Las instrucciones *exec* se utilizan para ceder la exploración de todas las cláusulas alternativas a partir de una dirección que llevan como parametro. Puesto que se cede más de una alternativa, los procesadores que reciben este trabajo crean los puntos de elección correspondiente durante la recomputación.

Instrucción try_exec

Funciona como *try_par*, excepto que tanto en modo LIBRE como DIRIGIDO crea un punto de elección.

```
procedure try_exec(n_var:integer; dir_exec:direccion);
begin
    PE := cima_stack();
    crear_punto_eleccion(PE);
   if modo = LIBRE then
   begin
       camino_exito[RE] := IC;
       RE := RE + 1;
       tabla_trabajos_pend[n_trabajos_pend].dir := PE;
       tabla_trabajos_pend[n_trabajos_pend].tipo := EXEC;
       tabla_trabajos_pend[n_trabajos_pend].salto := dir_exec;
       n_trabajos_pend := n_trabajos_pendientes + 1;
       enviar_aviso(controlador,trabajo_pendiente);
   end
   else /* modo DIRIGIDO */
   begin
       if camino_exito[RR] <> IC then
          IC := camino_exito[RR];
      else
      begin
          RR := RR + 1;
          if RR = RE then
             if conmu_Y = SEC then
                modo = LIBRE;
      end
   end
end
```

Figura 3.16: Instrucción try_exec

Instrucción retry_exec

Funciona como *retry_par*, excepto en que tanto en modo LIBRE como DIRIGIDO actualiza el punto de elección de la cima del *Stack*.

```
procedure retry_exec(dir_exec: direccion);
begin
    actualizar_punto_eleccion(IC);
   if mod = LIBRE then
   begin
       camino_exito[RE - 1] := IC;
       tabla_trabajos_pend[n_trabajos_pend].dir := PE;
       tabla_trabajos_pend[n_trabajos_pend].tipo := EXEC;
       tabla_trabajos_pend[n_trabajos_pend].salto := dir_exec;
       enviar_aviso(controlador,trabajo_pendiente);
   end
   else /* modo DIRIGIDO */
   begin
       RR := RR + 1;
       if RR = RE then
          if conmu_Y = SEC then
             modo = LIBRE;
   end
end
```

Figura 3.17: *Instrucción retry_exec*

3.4.6 Instrucciones modificadas para la explotación del Paralelismo_O

Se ha modificado el funcionamiento de las siguientes instrucciones de la WAM:

- tryMeElse
  En modo LIBRE crea un punto de elección como en la máquina secuencial, y copia la dirección de la instrucción en la pila de éxito, actualizando el correspondiente puntero y anotandolo en el punto de elección creado. En modo DIRIGIDO salta a la siguiente dirección del camino de éxito.

- retryMeElse
  En modo LIBRE actualiza el punto de elección con la dirección de la siguiente alternativa y la dirección de la cima de la pila de éxito con la dirección en que se encuentra. En modo DIRIGIDO no hace nada.

- trustMeElseFail
  En modo LIBRE elimina el punto de elección y actualiza la dirección de la cima de la pila de éxito. En modo DIRIGIDO no hace nada.

3.4.7 Instrucciones para la explotación del paralelismo O_bajo_Y

La explotación del paralelismo_Y puede optimizarse haciendo que las tareas_Y que comparten la ejecución de objetivos de la llamada paralela no reciban el camino de éxito completo si no hay paralelismo_O en el objetivo que van a ejecutar. Por esto se introduce una nueva instrucción que especifica si la ejecución de un objetivo tiene paralelismo_O:

call_o dir n_var

Esta instrucción funciona como *call* pero si la máquina está en estado de explotación del paralelismo_Y, anota en los objetivos que pone a disposición de los procesadores básicos desocupados si tienen paralelismo_O.

3.4.8 Instrucciones de PDP

El conjunto de instrucciones de PDP aparece en la tabla 3.1. Según estas instrucciones el código correspondiente al siguiente programa:

```
p(X,Y,Z) :- q(X,Y) & r(X,Z).

*q(X,Y) :- q1(X,Y).
*q(X,Y) :- q2(X,Y).
 q(X,Y) :- q3(X,Y).

*r(X,Y) :- r1(X,Y).
 r(X,Y) :- r2(X,Y).
        .
        .
        .
```

Instrucciones de la WAM				
putCon ri, c	bldCon c	getCon ri, c	uniCon c	exec
putVar ri, X	bldVar X	getVar ri, X	uniVar X	tryMeElse
putVal ri, X	bldVal X	getVal ri, X	uniVal X	retryMeElse
putStr ri, f	bldNil	getStr ri, f	uniNil	trustMeElseFail
putNil ri		getNil ri		switch_on_term
putLis ri		getLis ri		try
				retry
				trust
				alloc
				dealloc

Instrucciones con modo secuencial y paralelo
call
proceed

Instrucciones O paralelas
try_par
retry_par
try_exec
retry_exec

Instrucciones Y paralelas
par_exec
check_ground
check_independend
call_o
alloc_p
pop_goal
wait

Tabla 3.1: Tabla de instrucciones

y el objetivo:

$$:-p(a,Y,X).$$

es el siguiente:

```
entrada: put_con r0, a
         put_var r1, 1
         put_var r2, 2
         call p/3_0, 3
         stop

p/3_0:   alloc
         get_var r0, 0
         get_var r1, 1
         get_var r2, 2
         par_exec
         check_ground 0
         check_independent 1, 2
         alloc_p 2, n

         put_var 0
         put_val 1
         call_o p/2_0, 2

         put_val 0
         put_val 2
         call_o r/2_0, 2

         pop_goal
         wait
         proceed

 q\2_0:  try_par 0 q\2_1
         call q1
         proceed
 p\2_1:  retry_par q\2_2
         call q2
         proceed
 p\2_2:  trustMeElseFail
         call q2
         proceed

 r\2_0:  try_par 0 2\2_1
```

```
        call r1
        proceed
r\2_1:  trustMeElseFail
        call r2
        proceed
        .
        .
        .
```

En primer lugar apacece el código correspondiente al objetivo, que tras almacenar los argumentos en los registros llama al procedimiento p. Este procedimiento consta de una única cláusula cuyo cuerpo consiste en una llamada paralela. La instrucción *par_exec* conmuta entonces a modo *paralelo* en el que se comprueban las condiciones de cerrazón e independencia de las varibles de los objetivos. A continuación aparece el código correspondiente a cada objetivo. Se utiliza la instrucción *call_o* para indicar que estos objetivos presentan paralelismo_O. Finalmente aparece el código correspondiente a los objetivos q y r. Puesto que las cláusulas de estos objetivos están marcadas para ser ejecutadas en paralelo con la siguiente cláusula del procedimiento se utilizan las instrucciones *try_par*.

3.5 Procedimiento de backtracking

La Figura 3.18 muestra el algoritmo general de backtracking. Si el recorrido de la *pila de backtracking* nos lleva a un punto de elección probaremos las alternativas que tiene pendientes el procedimiento asociado, como ocurre en el caso secuencial. Un *Marcador de Objetivo Local* indica que todas las alternativas del objetivo ya han sido probadas (pués en otro caso se habrían encontrado puntos de elección en lugar del marcador) y por tanto tenemos que continuar haciendo backtracking. Un marcador de *Fin de llamada Paralela* indica que necesitamos nuevas soluciones de los objetivos de la llamada paralela. Por tanto buscaremos uno con alternativas pendientes que si es local se *reevalua*. Si es remoto, solo se pide su reevaluación al procesador que lo ejecutó si el fallo que se está tratando está *relacionado* con el objetivo. Si el marcador que se encuentra es una *LLamada Paralela*, nos indica que uno de los subobjetivos de la llamada a fallado y por tanto podemos interrumpir la computación de los objetivos que aún no hayan finalizado. Y por último si el marcador nos indica que es un *Objetivo Remoto* informamos al procesador padre del fallo producido.

```
procedure backtracking(P: programa; S:pila de backtracking; R:resolvente;
            v: sustitución);
var p(i) : lista de clausulas sin probar de un procedimiento;
begin
    if not vacio(S) then begin
        sacar(marcador,S);
        case marcador of
            procedimiento:
            begin
                sacar(R,S); sacar(v,S); sacar(p(a),S);
                resultado := ejecutar_objetivo(P,R[1],S,R,v,p(a)); end
            Objetivo Local:
            begin
                resultado := backtracking(P,S,R,v); end
            Fin Llamada paralela:
            begin
                i := numero_subobjetivo(Llamada_paralela);
                while i > 0 and not exito do begin
                    if estado(subobjetivo_i) = con_alt then begin
                        if remoto(subobjetivo_i) then
                            if relacionado(subobjetivo_i) then
                        resultado := pedir_reevaluar(subobjetivo_i,v);
                        else
                            resultado := reevaluar(subobjetivo_i,v);
                    if resultado = exito then
                        for j := i to numero_subobjetivos do begin
                            buscar_procedimiento(subobjetivo_j,P,p(a));
                            resultado := ejecutar_objetivo(P,subobjetivo_j,S,R,v,p(a));
                        end
                    i := i - 1;
                    end
                end
                if not exito then resultado := backtracking(P,S,R,v);
            end
            Llamada paralela:
            begin
                liberar_tareas(Llamada_paralella);
                resultado := backtracking(P,S,R,v); end
            Objetivo Remoto:
            begin
                enviar_fallo(procesador_padre);
                S := ∅;
                R := []; end
        end
        else
            resultado := fallo;
    return resultado
end
```

Figura 3.18: Algorítmo general de backtracking

3.6 Intercambio de trabajo

La creación de una nueva tarea consiste en la asignación del trabajo correspondiente a la tarea a un procesador desocupado. Los *procesadores básicos* envían los trabajos que tienen pendientes al procesador que les indica el controlador. Los procedimientos de envío de trabajo se encargan de seleccionar la información necesaria para realizar los trabajos pendientes.

3.6.1 Procedimiento de envío de una tarea_Y

Este procedimiento envía un objetivo paralelo, anotando la espera de su respuesta. El envío incluye la dirección del Marcador de la llamada Paralela correspondiente y el orden del objetivo en la llamada paralela (*slot*), de forma que cuando se reciba la respuesta se pueda actualizar la llamada paralela. Se envía también la dirección del procedimiento asociado al objetivo y la vinculación de sus variables. Si la tarea que se va a crear es primaria, se envía el camino de éxito completo, si es secundaria se envía únicamente la parte del camino correspondiente al objetivo paralelo si existe.

3.6.2 Procedimiento de envío de una tarea_O

El procedimiento de envío correspondiente a una tarea_O consiste en enviar a un procesador desocupado un camino del árbol de búsqueda a seguir. Si la tarea que se va a crear es primaria se envía el camino de éxito de la tarea actual cambiando la última dirección por la de la siguiente cláusula inexplorada, que está contenida en el *punto de elección* asociado. Si la tarea que se va a crear es secundaria, es decir, el paralelismo_O ha surgido durante la ejecución de un objetivo paralelo, se realiza el producto cruzado, aplicando la regla de combinación: se envía el mismo camino de la tarea padre para los objetivos paralelos a la izquierda del objetivo de procedencia, la siguiente rama a probar para el objetivo de procedencia, y todas las ramas para los objetivos a la derecha del de procedencia.

```
procedure enviar_tarea_Y(M:mensaje; Obj: objetivo; S: slot;
        LP:marcador_llamada_paralela; MPC: Marcador_producto_cruzado;
        C: camino_exito; T: tipo_tarea);
begin
var v:dato;
    incluir_mensaje(M, dir(LP));
    incluir_mensaje(M, S);
    incluir_mensaje(M, dir(Obj));
    for i := 1 to num_var(Obj) do
        v := desref(var(Obj,i));
        incluir_mensaje(M, v);
    end
    if T = PRIMARIA then
        incluir_mensaje(M, C);
    else begin
        C = C[MPC[Obj]];
        incluir_mensaje(M, C);
    end
    /* se actualiza el LP */
    num_respuestas_esperadas(LP) := num_respuestas_esperas + 1;
end
```

Figura 3.19: Procedimiento de envio de una tarea_Y

```
procedure enviar_tarea_O(M:mensaje; PE: punto_elección
        C: camino_exito);
begin
var longitud: integer;
    longitud := longitud(C);
    C[longitud - 1] := siguiente_dir(PE);
    incluir_mensaje(M, C);
end
```

Figura 3.20: Procedimiento de envío de una tarea_O primaria

```
procedure enviar_tarea_O_secundaria(M:mensaje; PE: punto_elección
        MPC: marcador_producto_cruzado; C: camino_exito);
begin
var longitud, longitud_llamada: integer;
        objetivo_procedencia: integer;
        C_auxiliar: camino_exito;
    longitud := longitud(C);
    longitud_llamada := longitud(MPC);
    C[longitud - longitud_llamada] := siguiente_dir(PE);
    incluir_mensaje(M, C_auxiliar);
    objetivo_procedencia := objetivo_procedencia(MPC);
    for i:= 1 to num_objetivos(MPC) do
        if i < objetivo_procedencia then
        begin
            C_auxiliar := C[MPC[i]];
            incluir_mensaje(M, C_auxiliar);
        end
        elseif i = objetivo_procedencia then
        begin
            C_auxiliar := C[MPC[i]];
            longitud := longitud(C_auxiliar[MPC[i]]);
            C_auxiliar[longitud - 1] := siguiente_dir(PE);
            incluir_mensaje(M, C_auxiliar);
        end
        elseif i > objetivo_procedencia then
        begin
            C_auxiliar := ∅
            incluir_mensaje(M, C_auxiliar);
        end
    end
end
```

Figura 3.21: Procedimiento de envío de una tarea_O secundaria

4

Planificación del trabajo en PDP

4.1 Introducción

La planificación del trabajo pendiente sobre los procesadores de un sistema paralelo es un factor decisivo en el rendimiento. La planificación puede ser *estática* o *dinámica*, dependiendo del tiempo en que se realiza. En la primera, todas las tareas de una computación se fijan *a priori* y se asignan estáticamente a los procesadores de la red antes de comenzar la ejecución. Para programas Prolog, que incluyen el indeterminismo en su proceso de ejecución, es conveniente aplicar una planificación dinámica que asigne los procesadores según aparecen nuevas tareas. El control de la planificación dinámica puede ser *distribuido* o *no-distribuido* dependiendo de que las decisiones de planificación se distribuyan o no entre diferentes procesadores. Los algoritmos distribuidos de planificación son habituales en sistemas con total o parcial memoria compartida en los que cada procesador puede acceder directamente a la información de los restantes. Así, el sistema MUSE [2] coloca en una zona de memoria compartida una copia de las estructuras que contienen la información sobre las tareas pendientes. En los sistemas con memoria completamente distribuida [75] [19] [11] es habitual centralizar la información para evitar la multiplicidad de mensajes que informan sobre el estado de un mismo procesador.

PDP funciona bajo un control jerárquico realizado por los *controladores* que registran la información sobre la carga de trabajo de los procesadores básicos, la analizan y aplican una política de planificación concebida para que el tráfico de mensajes sea mínimo. Las tareas paralelas que ceden los

procesadores básicos viajan del origen al destino indicado por el controlador, pero sin pasar a través de este. Cada controlador está asociado a un grupo de procesadores (Figura 4.1). El asignación de trabajo a procesadores pertenencientes a distintos grupos se realiza bajo condiciones más fuertes que en el caso de procesadores del mismo grupo.

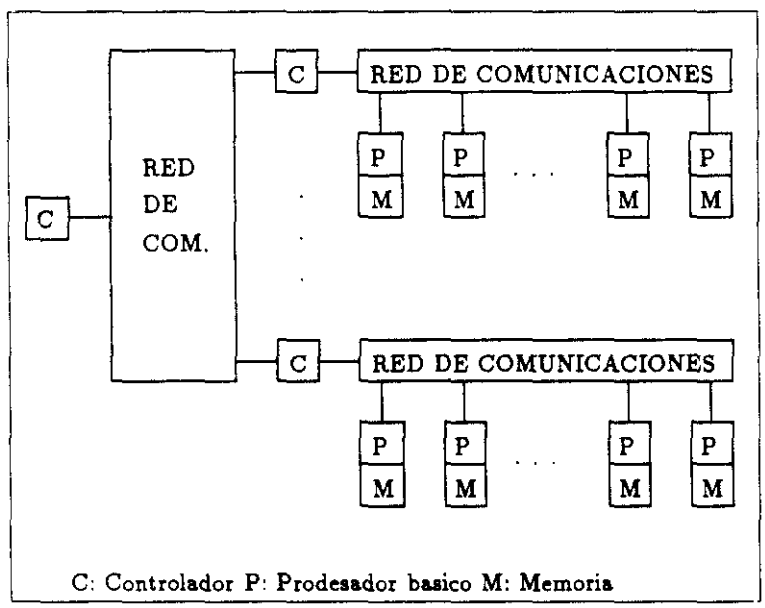


Figura 4.1: Organización del sistema

La distribución de trabajo entre los procesadores del sistema presenta distintos aspectos:

- **Establecimiento de las unidades de división de los trabajo.** Cuando un procesador tiene pendiente la ejecución de una llamada paralela es necesario establecer un reparto de los objetivos independientes entre el procesador padre y los restantes procesadores. De la misma forma, cuando un procesador dispone de distintos caminos alternativos anotados con paralelismo para ejecutar un objetivo, se fija un reparto de estos caminos entre el procesador padre y los restantes.
- **Selección de los trabajos a compartir.**

Los procesadores básicos se encargan de seleccionar los trabajos que se ejecutan en paralelo teniendo en cuenta estimaciones sobre la granularidad del trabajo.

- **Asignación de los trabajos seleccionados a los procesadores.** Los controladores se encargan de asignar a los procesadores desocupados las tareas seleccionadas para ser ejecutadas en paralelo. Para realizar la selección tienen en cuenta una serie de factores (orden de explotación de los distintos tipos de paralelismo, proximidad topológica, balance de carga, antiguedad de los trabajo).

En este capítulo se describen cada uno de estos aspectos.

4.2 Unidades de división del trabajo

En el caso de la explotación del paralelismo_Y el procesador padre comparte la ejecución de la llamada paralela con otros procesadores. La ejecución de cada uno de los objetivos independientes de la llamada constituye una tarea_Y. PDP asigna cada una de estas tareas a un procesador desocupado, de forma que son ejecutadas por el procesador padre si no hay procesadores desocupados. Un reparto equitativo de las tareas_Y entre el procesador padre y otro procesador, de forma que a su vez este compartiese su trabajo con otros procesadores, complicaría el control de la llamada paralela ya que la ejecución de las tareas_Y debe devolver una respuesta al procesador padre.

En el caso de la explotación del paralelismo_O, el procesador padre cede las ramas inexploradas del árbol de búsqueda de una en una (Figura 4.2). De esta forma el procesador que recibe el trabajo no necesita crear puntos de elección durante la recomputación, puesto que las alternativas que quedan pendientes no le pertenecen. Así se optimiza tanto el tiempo como el espacio durante la recomputación.

Otras estrategias posibles de división del trabajo son las siguientes:

- **El procesador padre cede todas las alternativas excepto una** produciendose el reparto de la Figura 4.3.

- **Las alternativas se reparten de forma equitativa** mediante las instrucciones *_exec* (Figura 4.4).

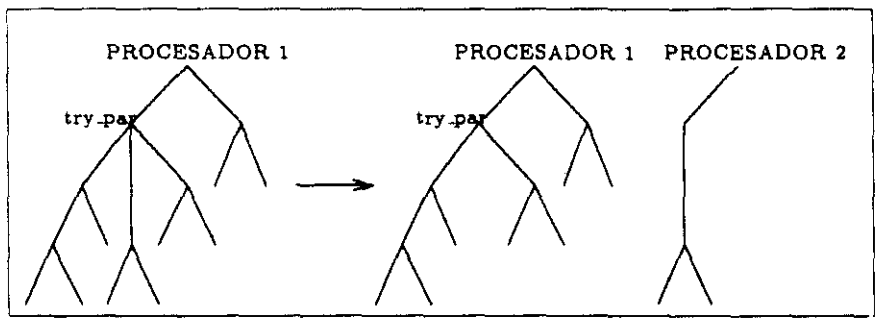


Figura 4.2: Reparto 1: El procesador padre cede un única alternativa cada vez.

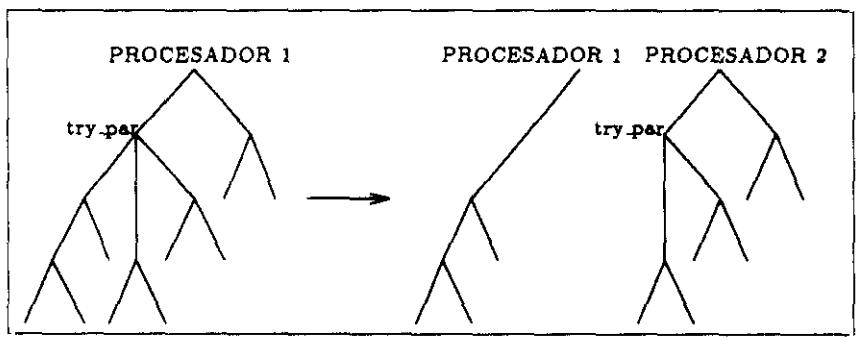


Figura 4.3: Reparto 2: El procesador padre cede todas las alternativas excepto una.

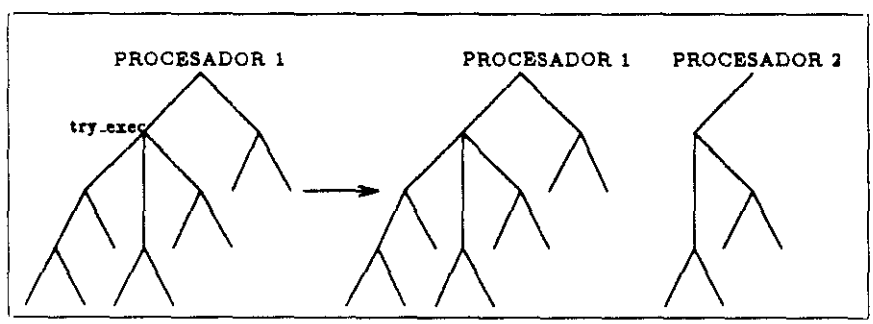


Figura 4.4: Reparto 3

Las tres estrategias han sido implementadas para la explotación del paralelismo_O, obteniendo los mejores resultados cuando el procesador padre se queda con todas las alternativas excepto una. También se ha estudiado el comportamiento cuando el paralelismo_O se explota siguiendo un modelo por *copia*, resultando que no hay diferencias apreciables. La segunda estrategia es ligeramente mejor ya que el único recargo que se introduce es actualizar el registro *back_point*, que marca el punto de la pila de control hasta el que se pueden tomar alternativas pendientes. Si se optase por una de las otras estrategias tendría que actualizarse el punto de elección correspondiente para dejar constancia de las alternativas cedidas.

4.3 Selección en los procesadores básicos de los trabajos a compartir: Granularidad

Los factores determinantes para la ejecución paralela de una tarea son la **independencia** y la **granularidad**. En el caso del paralelismo_Y la independencia de los objetivos que constituyen las tareas_Y puede ser incondicional o condicionada a comprobaciones que se realizan en tiempo de ejecución sobre la cerrazón e independencia de los argumentos de los objetivos. Por lo tanto, la condición de independencia se cumple si una llamada paralela aparece anotada y las comprobaciones tienen éxito (en caso de ser condicional). En el caso del paralelismo_O, la condición de independencia se cumple siempre entre las tareas_O. La selección de las tareas entre aquellas que cumplen el prerequisito de independencia se realiza dependiendo de la granularidad, es decir, el recargo de trabajo que tiene asociado la ejecución paralela de una tarea debido a la gestión del paralelismo: creación, comunicación y planificación de tareas. Es por tanto fundamental tener una estimación del *grano* de un trabajo, es decir, de su carga computacional, para saber si el ahorro de tiempo de su ejecución paralela supera el tiempo introducido por la gestión del paralelismo.

La granularidad se puede estimar en tiempo de compilación, en tiempo de ejecución o repartirse entre ambos. Los métodos que se aplican en tiempo de compilación [25, 64] estiman la granularidad analizando estáticamente la estructura del programa. Sin embargo, la granularidad de muchas tareas depende de parametros que solo se conocen en tiempo de ejecución (datos de entrada) y por tanto los métodos realizados en tiempo de compilación pueden resultar poco precisos. Por el contrario, los métodos que estiman la granu-

laridad en tiempo de ejecución pueden realizar estimaciones precisas, pero introducen un recargo en el tiempo de ejecución que impide la mejora del rendimiento. PDP sigue un esquema que obtiene toda la información posible en tiempo de compilación, de manera que la evaluación de la granularidad en tiempo de ejecución supone un recargo muy pequeño. Este enfoque ha sido seguido por [64], [25] y [50]. En [64] Tick propone estimar la granularidad de un procedimiento como la suma de las granularidades de los procedimientos de sus llamadas, y considerar la granularidad de las cláusulas autorecursivas con valor uno. De esta forma las llamadas a s procedimientos se anotan en tiempo de compilación con el peso correspondiente, y un sencillo mecanismo en tiempo de ejecución toma las decisiones de planificación en base a estos pesos. En [78] se mejora el método estimando solo la granularidad relativa de los objetivos. Debray *et al.* [25] [26] presentan un metodo en tiempo de compilación para obtener la granularidad de los predicados. En primer lugar se derivan relaciones entre los tamaños de los argumentos de entrada y de salida, para después establecer ecuaciones de recurrencia de las funciones que dan la estimación de la granularidad de los predicados. Estas ecuaciones se resuelven en tiempo de compilación derivando funciones de granularidad de los predicados en función de los tamaños de los argumentos de entrada. De esta forma se consiguen estimaciones de alta precisión, aunque el recargo en tiempo de ejecución debido a la obtención del tamaño de los argumentos puede ser considerable, y además no siempre se dispone de un método sistemático para resolver las ecuaciones de recurrencia. En [50] se propone una optimización de este método basada en la observación de que a menudo los términos han sido recorridos antes de necesitar conocer su tamaño.

Estas ideas que han dado lugar al sistema CASLOG [27] han sido incorporadas a PDP aunque el recargo en tiempo de ejecución se reduce aplicando el analisis en el momento en que se conoce el tamaño de los términos que intervienen en la función que dá la granularidad. Se aplican además otros controles de granularidad basados en observaciones heurísticas.

4.3.1 Control de la granularidad en tiempo de compilación

El esquema de estimación de granularidad propuesto en [26] acota la granularidad de una cláusula por la granularidad de la unificación de la cabeza junto con la granularidad de cada uno de los objetivos del cuerpo. Debido al indeterminismo, el número de veces que un objetivo es ejecutado depende del número de soluciones que los objetivos que le preceden pueden generar.

Por lo tanto para la cláusula

$$H : -L_1, ..., L_m$$

si el tamaño de la entrada del literal L_i es T_{L_i}, y el número de soluciones que L_i puede generar es Num_{L_i} entonces la granularidad de la cláusula C puede expresarse como

$$G_C \leq \tau + \sum_{i=1}^{m}(\prod_{j \preceq i} Num_{L_j}(T_{L_j})G_{L_i}(T_{L_i})$$

donde τ es el tiempo necesario para resolver la cabeza de la cláusula. En la fórmula $j \preceq i$ denota que L_j precede a L_i en el grafo de dependencia que representa las relaciones de llamada entre los predicados de la cláusula. El valor de τ depende de la métrica usada como unidad para la medida de la granularidad, si es el número de resoluciones, τ es 1, si el número de unificaciones, τ es la aridad de H. A partir de las ecuaciones que dan la granularidad de las cláusulas se obtiene la ecuación que dá la granularidad del predicado. En [26] se presenta un algoritmo para obtener una estimación del número de soluciones, usando propiedades de la unificación para estimar el número de posibles vinculaciones de los conjuntos de variables. El número de soluciones posibles de una cláusula es entonces el número de soluciones posibles del conjunto de variables que aparecen en su cabeza. Los objetivos recursivos se manejan usando expresiones simbólicas para denotar el número de soluciones que generan y resolviendo (o estimando una cota superior) la ecuación en diferencias resultante. Existen algunos trabajos sobre la resolución automática de ecuaciones en diferencias [23, 47]. Siempre es posible reducir un sistema lineal de ecuaciones en diferencias a una única ecuación lineal de una variable. Los programas de [23, 47] resuelven una clase de ecuaciones lineales en diferencias con coeficientes constantes. El problema de resolver ecuaciones en diferencias lineales con coeficientes polinomiales también puede reducirse a la resolución de ecuaciones en diferencias ordinarias. Aunque en general las ecuaciones en diferencias no lineales son mucho más complejas que las lineales, algunas de ellas pueden convertirse en lineales por transformación de variables.

Este análisis de granularidad es aplicado por el sistema CASLOG [27] para obtener una estimación de la granularidad en forma de expresión algebráica del tiempo de ejecución de un predicado p para una entrada de

tamaño n: $T_q(n)$. Esta expresión se evalua en tiempo de ejecución, cuando se conoce el tamaño de la entrada, proporcionando una estimación del trabajo pendiente en la llamada a un predicado. Por ejemplo, supongamos que para el predicado

```
p([]).
p([H|L]) :- q(H), p(L).
```

los literales $q(H)$ y $p(L)$ del cuerpo de la segunda cláusula sean independientes, y por tanto candidatos para ser ejecutados en paralelo. Supongamos que la expresión $T_q(n)$ que dá la granularidad de q para una entrada de tamaño n es $3n^2$ y que el coste de creación de una tarea paralela es de 48 unidades de tiempo. Entonces, se puede generar un código para la segunda cláusula cuya ejecución paralela esté condicionada a la estimación de granularidad:

n := size(H);
if $3n^2 < 48$ **then** execute p and q secuencialmente
else ejecutar q y p en paralelo

Esta expresión algebráica de la granularidad proporcionada por CASLOG se incorpora en tiempo de compilación al código de entrada a PDP. Para incluir estas expresiones en los programas de PDP se añade una nueva instruccion

check_granu *tamaño*

que precede a las instrucciones *call* de una llamada paralela. Si no se cumple la condición de granularidad se conmuta a modo de ejecución secuencial (conmu_Y = SEC). Esta instrucción es opcional. El parametro *tamaño* es el resultado de despejar n de la expresión de granularidad. En el ejemplo si $3n^2 < 48$ entonces $n < 4$, luego el parametro tamaño tomaría el valor 4.

4.3.2 Control de la granularidad en tiempo de ejecución

Aplicar la estimación de la granularidad que proporciona CASLOG para un predicado requiere conocer el tamaño de los datos del predicado. Este cálculo puede ser muy complejo en algunos casos por lo que la estimación de la granularidad a partir de él puede incrementar considerablemente el

tiempo introducido por la gestión del paralelismo. Al explotar paralelismo_Y en PDP hay un punto en el que el tamaño del dato es conocido: en el envío del entorno cerrado correspondiente a un objetivo. En este punto se prepara un mensaje con el objetivo y los términos desreferenciados a que estan vinculadas sus variables. PDP aplica el control de granularidad basado en la estimación proporcionada por CASLOG en este punto, aprovechando la creación del mensaje para medir el tamaño del dato. Si el grano es inferior al coste de creación de la tarea paralela correspondiente, la taréa se ejecuta secuencialmente. En la tabla 4.1 aparecen los programas de prueba (*benchmarks*), utilizados para la evaluación del paralelismo_Y y el control de granularidad descrito en esta sección. La tabla 4.2 presenta la mejora del rendimiento obtenida al aplicar esta técnica (sin g.: sin control de granularidad, con g.: con control de granularidad). Con 4 procesadores no se aprecia su efecto, ya que el propio tamaño del sistema limita la granularidad. El efecto más importante se aprecia con 8 procesadores, para los programas *merge* y *qsort*, ya que los tamaños de los datos del programa *matriz* son mucho menores que en los anteriores. Con 16 transputers el efecto tampoco es apreciable, debido a que no hay suficiente paralelismo en el programa para ocupar a todos los procesadores del sistema, y el rendimiento no mejora a partir de 10 procesadores.

Programa	Descripción
mergesort	Ordena una lista de números por mezclas
quicksort	Ordena una lista de números por quicksort
matriz	Multiplica una matriz por un vector

Tabla 4.1: Programas de prueba para la evaluación del paralelismo_Y

4.3.3 Estimación heurística de la granularidad

PDP aplica otros controles de granularidad basados en la observación del comportamiento seguido por grandes grupos de programas Prolog. Estos controles heurísticos producen una respuesta en un tiempo muy breve ya que únicamente requieren la comprobación del parametro considerado y no incrementan el tiempo de ejecución.

	4 proc.		8 proc.		16 proc.	
programa	sin g.	con g.	sin g.	con g.	sin g.	con g.
merge(500)	1.38	1.4	1.78	1.8	1.93	1.94
qsort(700)	1.42	1.42	1.91	2.25	2.25	2.25
matriz(75)	1.38	1.38	1.73	1.73	1.78	1.78

Tabla 4.2: Control de la granularidad por el tamaño del dato

Estimación heurística de la granularidad en la explotación del paralelismo_O

Basandonos en la observación de que la obtención de las distintas soluciones a un objetivo producen pilas de un tamaño similar, podemos hacer una estimación del tiempo que tardará un procesador en alcanzar la siguiente solución. En la tabla 4.3 aparece la descripción de los programas de prueba utilizados para la evaluación del paralelismo_O y el control de granularidad descrito. En la tabla 4.4, que presenta el tamaño de la pila de control al alcanzar las primeras soluciones de estos programas, se puede observar la próximidad de estos valores.

Programa	Descripción
queen	Colocación de N reinas en un tablero de ajedrez NxN de forma que no se ataquen
chat	Base de datos geográfica
farmer	Acertijo
query	Pequeña base de datos [12]
salt	Acertijo
mm	Juego del mastermind

Tabla 4.3: Programas de prueba para la evaluación del paralelismo_O

Cuando un procesador básico obtiene su primera solución registra el tamaño de sus pilas. Cada trabajo está asociado a un punto de elección de la pila, de manera que antes de cederlo a otro procesador se comprueba si el tamaño del tramo de pila entre dicho punto de elección y el punto

programa	solución 1	solución 2	solución 3	solución 4
queen6	640	640	640	640
queen8	1061	1061	1061	1061
chat	400	402	382	384
farmer	658	647		
query	81	81	81	81
salt	365	365	333	297
mm	6121	6093	6101	6065

Tabla 4.4: Tamaño de la pila de control al alcanzar las soluciones

alcanzado cuando se obtuvo la última solución supera cierto valor umbral ajustado experimentalmente. El procedimiento no introduce recargo ya que solo requiere una anotación al alcanzar cada solución y una comparación antes de ceder un trabajo.

La Figura 4.5 representa una posible situación de la pila de un procesador básico. $S1$ es el tamaño alcanzado por la pila cuando se obtuvo la útima solución. $T1$ y $T2$ son los puntos de la pila en que se encuentra el punto de elección asociado a dos tareas pendientes. Se observa que $S1$ está muy próximo a $T2$ y por tanto la diferencia $L2$ entre estos puntos será menor que el valor umbral. Esto quiere decir, como vemos por el árbol de búsqueda, que $T2$ tiene un grano pequeño y no es cedido para ser ejecutado en otro procesador. Por el contrario, $T1$,que se encuentra en la base de la pila, está asociado a una tarea de alta granularidad como se ve en el árbol de búsqueda. La distancia $L1$ entre $T1$ y $S1$ será mayor que el valor umbral y por tanto $T1$ se cederá a otro procesador.

Se han realizado diversas medidas para evaluar el valor umbral, que depende del programa considerado y del número de procesadores del sistema. Para los programas de prueba considerados y con 8 y 16 procesadores el valor umbral se encuentra entre *tamaño_pila*/6 y *tamaño_pila*/7, donde *tamaño_pila* es el valor de la cima de la pila de control cuando se alcanzó la última solución, dandose la mayor desviación para el programa *zebra*, que obtiene los mejores resultados con un valor umbral de *tamaño_pila*/4. La mejora del rendimiento conseguida al introducir este mecanismo aparece en la Tabla 4.5. Con 4 procesadores el control introducido no tiene efecto. Con 8 procesadores mejora ligeramente el rendimiento y el mayor efecto

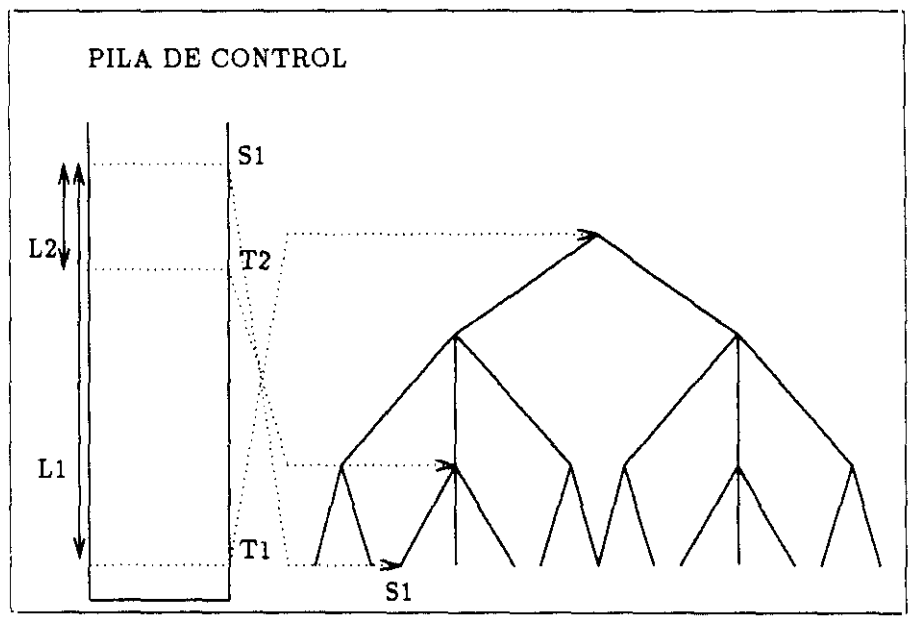


Figura 4.5: Estimación de la granularidad del paralelismo O

se consigue con 16 procesadores. Esta técnica es más efectiva al aumentar el tamaño del sistema, ya que si es pequeño el propio tamaño limita la movilidad del trabajo.

	4 proc.		8 proc.		16 proc.	
programa	sin g.	con g.	sin g.	con g.	sin g.	con g.
query	2.6	2.6	2.8	2.9	3.0	3.4
zebra	1.8	1.8	3.5	3.7	3.6	3.9
mm	2.1	2.1	3.2	3.4	4.1	4.5
queen(8)	2.9	2.8	7.0	7.3	12.3	12.9
queen(9)	2.4	2.4	7.5	7.7	13.8	14.2
queen(10)	3.0	3.1	7.8	7.9	14.5	14.7

Tabla 4.5: Control heurístico de la granularidad en la explotación del Paralelismo_O

Estimación heurística de la granularidad en la explotación del paralelismo_Y

Otro factor que puede aprovecharse para controlar la granularidad se basa en la observación de que en muchos programas los objetivos de mayor grano se encuentran cerca de la raíz del árbol de búsqueda. Por tanto podemos estimar que los sucesivos trabajos pendientes que se producen tienen un grano igual o inferior al de los anteriores. Basandonos en esta observación podemos optimizar el rendimiento si cada procesador básico deja de explotar el paralelismo_Y cuando en la última ocasión en que lo ha hecho ha tenido que esperar por la respuesta a un objetivo un tiempo mayor que el que ha tardado en ejecutarse dicho objetivo. Se trata de un mécanismo que no introduce recargo en el tiempo de ejecución, ya que únicamente exige medir los tiempo de espera y de ejecución del objetivo. Este último se envía al procesador padre junto con el objetivo. La Tabla 4.6 muestra el resultado de incluir este control en el sistema. En la tabla se observa que la aplicación de este control de granularidad no tiene efecto para 4 transputers, empieza a ser efectivo con 8 transputers y proporciona los mejores resultados con 16 transputers.

	4 proc.		8 proc.		16 proc.	
programa	sin g.	con g.	sin g.	con g.	sin g.	con g.
merge(500)	1.38	1.39	1.78	1.8	1.93	1.95
qsort(700)	1.42	1.42	1.91	2.0	2.25	2.29
matriz(75)	1.38	1.38	1.73	1.74	1.78	1.8

Tabla 4.6: Control heurístico de la granularidad en la explotación del Paralelismo_Y

4.4 Planificación del trabajo por los Controladores

Los controladores son los encargados de registrar los trabajos paralelos y gestionar la planificación. Para ello reciben información sobre el estado de los procesadores básicos. En relación con la carga de trabajo un procesador básico puede estar en los siguientes estados: *desocupado*, si no tiene trabajo, *ocupado* si está realizando un trabajo, o *donante*, si tiene trabajo pendiente. PDP no sigue uno de los algoritmos clásicos de reparto de trabajo [42, 62, 76] en los que la donación del trabajo es iniciada bien por el donante o por el receptor sino que tanto los procesadores donantes, como los desocupados, informan de su estado al controlador encargado del grupo. El controlador indica a los procesadores básicos donantes a que procesador desocupado deben enviar su trabajo pendiente. De esta forma se evita que procesadores ocupados sean interrumpidos por peticiones que no pueden atender. Asímismo se reduce el número de mensajes intercambiados. El esquema seguido para la asignación de trabajos es el siguiente:

- Un procesador que tiene trabajo pendiente hace una estimación del grano de este trabajo como se describe en la Sección 4.3. Si el grano es suficiente avisa de ello al controlador, indicando el tipo de paralelismo al que corresponde el trabajo.

- Los controladores reciben avisos de procesadores donantes así como de procesadores desocupados. Al recibir un aviso de un procesador desocupado selecciona un trabajo pendiente de acuerdo a los criterios que se desarrollan en el apartado 4.4.4.

- Si un controlador comprueba que todos los procesadores de su grupo están ocupados, y quedan trabajos pendientes, informa de ello al con-

troladores del nivel superior. Los controladores que comprueban que el porcentaje de procesadores desocupados en su grupo excede un valor fijado experimentalmente informan al controlador superior de que pueden recibir trabajo.

Aunque el envío de información al controlador no es exactamente periódico, está sujeto a restricciones que evitan que se informe de cada cambio en la carga de trabajo de un procesador, como se describe en la Figura 4.6. Mantener el número exacto de trabajos pendientes que tiene cada procesador requiere un envío continuo de mensajes con el controlador, que incrementa considerablemente el tiempo de ejecución. Por esto se mantiene una aproximación a este valor. Los procesadores básicos sólo informan al controlador cuando pasan de tener trabajos pendientes a no tenerlos y viceversa. De esta forma el controlador sabe qué procesadores son donantes y cuales están desocupados, aunque desconoce la cantidad de trabajo pendiente.

```
case of
    trabajo_pendiente:
        num_trabajo_pendientes := num_trabajos_pendientes + 1;
        if num_trabajos_pend = 1 then
            enviar_aviso_tra(num_trabajos_pendientes);
    peticion_trabajo:
        enviar_trabajo();
        num_trabajo_pendientes := num_trabajos_pendientes - 1;
        if num_trabajos_pend = 0 then
            enviar_aviso_notra();
    ejecucion_trabajo_pendiente:
        num_trabajo_pendientes := num_trabajos_pendientes - 1;
        if num_trabajos_pend = 0 then
            enviar_aviso_notra();
end
```

Figura 4.6: Criterio de los procesadores básicos para actualizar la información sobre su estado

4.4.1 Controladores

El esquema de funcionamiento del controlador aparece en la Figura 4.7. Uno de estos procesadores está identificado como inicial y comienza la eje-

```
type
    t_trab: array[1..NMAX_TRA] of integer; t_pet: array[1..NMAX_PET] of integer;
process controlador(p:programa; num_procesadores:integer);
var
    resultado: boolean; trabajo, peticion: integer;
    num_trabajos, num_peticiones, num_pet_encurso: integer;
    tabla_trabajos: t_trab; tabla_peticiones, tabla_pet_encurso: t_pet;
begin
    while true do begin
       recibir_mensaje(M);
      case tipo(M) of
          trabajo_pendiente:
             if (num_peticiones > 0) then seleccionar_procesador(origen(M),
                          tabla_peticiones, peticion, num_peticiones);
             if (peticion <> 0) then begin
                enviar_peticion(origen(M));
                marcar_peticion(peticion); num_peticiones := num_peticiones - 1;
                num_pet_encurso := num_pet_encurso + 1; end
             else begin
                anotar_trabajo(origen(M), tabla_trabajos);
                num_trabajos := num_trabajos + 1; end
          petición_trabajo:
             if (num_trabajos_pend > 0) then
                if (origen(M) <> controlador_padre) or (num_peticiones = 0) then
                   seleccionar_trabajo(origen(M),tabla_trabajos,trabajo,num_trabajos);
             if (trabajo <> 0) then begin
                enviar_peticion(origen(tabla_trabajo[trabajo]));
                marcar_peticion(peticion); num_peticiones := num_peticiones - 1;
                num_pet_encurso := num_pet_encurso + 1; end;
             else begin
                anotar_peticion(origen(M), tabla_peticiones);
                num_peticiones := num_peticiones + 1; end
          borrar_trabajo:
             borrar_trabajo_pendiente(origen(M));
          confirmación:
             borrar_peticion(origen(M)); borrar_trabajo(origen(M));
             num_pet_encurso := num_pet_encurso - 1;
          negacion:
             desmarcar_peticion(origen(M)); borrar_trabajo(trabajo(M));
             num_peticiones := num_peticiones + 1;
             num_pet_encurso := num_pet_encurso - 1; end
       end
       if (num_peticiones > MIN_OCU) then /* pocos ocupados */
          enviar_peticion_controlador(controlador_padre);
    end
end
```

Figura 4.7: Esquema del controlador

cución del objetivo mientras los restantes procesadores envian peticiones de trabajo al controlador que las anota en una tabla (*anotar_petición*). Cuando surge paralelismo en una ejecucion, el procesador informa de ello al controlador, que también lo anota (*anotar_trabajo*). Al recibir un aviso de trabajo pendiente el controlador comprueba si hay procesadores desocupados (*num_peticiones* > *0*) y si es así *selecciona* uno de estos procesadores de acuerdo a los criterios de planificación que se presentan en el apartado 4.4.2, enviando la petición al procesador que posee el trabajo. Cuando el controlador recibe una petición de trabajo, comprueba si hay trabajos pendientes (*num_trabajos* > *0*) enviando la petición al procesador con trabajo pendiente seleccionado. Si la petición de trabajo viene de un controlador de nivel superior, solo es atendida si todos los procesadores del grupo estan *ocupados*. El controlador *marca* las peticiones pendientes de ser confirmadas. El procesador que recibe un trabajo envía *confirmación* al controlador. Puede ocurrir que al llegar el mensaje de petición a un procesador, el trabajo ya no esté disponible. En este caso el procesador envía un mensaje de *negación* al controlador. Cuando el controlador recibe un mensaje de confirmación borra de sus tablas el trabajo y la petición pendientes, mientras que si recibe un mensaje de negación, borra el trabajo y *desmarca* la petición, para que vuelva a ser considerada cuando haya trabajos pendientes. Cuando un procesador empieza a ejecutar un trabajo que el controlador tenía anotado como pendiente, envía un mensaje de *borrar_trabajo* al controlador. Si el controlador detecta que el número de procesadores desocupados de su grupo supera un umbral, (MIN_OCU), que depende del número de procesadores por grupo y de grupos del sistema, envía una petición al controlador del nivel superior.

4.4.2 Planificación del trabajo por los controladores

Los controladores realizan la planificación atendiendo en primer lugar a una **restricción** que existe en la asignación de las tareas_Y para que el resultado de la ejecución sea correcto: los objetivos ejecutados en un mismo procesador deben mantener el orden o precedencia dado por un recorrido en profundidad y de izquierda a derecha del árbol de búsqueda. Para la asignación a procesadores desocupados de las tareas_Y que cumplen la restricción y las tareas_O, los controladores aplican una política de planificación que considera diversos factores.

4.4.3 Restricción a la planificación: Precedencia de objetivos

Al explotar el paralelismo_Y, si no se mantiene la precedencia entre los objetivos ejecutados por una misma tarea, es decir si se ejecutan en un orden distinto al que dá un recorrido en profundidad y de izquierda a derecha del árbol de búsqueda, pueden aparecer los problemas *garbage slot* y *trapped goal*, señalados por Hermenegildo [33]. El primero se refiere a la aparición de una zona de memoria correspondiente a un objetivo A cedido por otro procesador que se encuentra protegida por el entorno correspondiente a otro objetivo B anterior en el árbol. Al tratar un fallo puede ocurrir que se produzca *backtracking* de A antes que de B. Si como resultado de dicho backtracking, el entorno correspondiente a A debe desaparecer, quedará una zona de la pila vacía. Los *garbage slots* de la memoria pueden recuperarse en un proceso de recolección de basura y en muchos casos se recuperan directamente por un backtracking posterior. El problema llamado *trapped goal* es más grave, ya puede llevar a un funcionamiento erroneo del sistema. Este problema aparece cuando un objetivo que se encuentra en la memoria "tapado" por uno anterior requiere la obtención de nuevas soluciones. En general el espacio reservado para el cálculo de la primera puede ser insuficiente para los cálculos posteriores.

Para preservar la precedencia de objetivos, el controlador almacena la historia de las tareas_Y realizadas por cada procesador, comprobando que se cumple la precedencia a partir de un número identificativo asignado a cada tarea_Y y del nivel que corresponde a cada tarea en el árbol de búsqueda, que se calcula sumando 1 al nivel del objetivo padre. El controlador sólo dá una orden de asignación de una tarea_Y cuando se trata de una tarea de nivel inferior a la última ejecutada en el procesador destino, o que siendo del mismo nivel tiene un identificativo mayor.

4.4.4 Criterios de planificación

Para establecer la política de planificación se ha considerado en primer lugar el caso en existen tareas_Y y tareas_O pendientes. Una vez establecido un orden de explotación entre los dos tipos de tareas se ha realizado una selección entre distintas estrategias de planificación basadas en diferentes critérios: proximidad topológica entre el procesador donante y receptor, balance de carga entre los procesadores donantes y antiguedad de los trabajos pendientes.

Orden de explotación de los distintos tipos de tareas

Se ha realizado un estudio comparando distintas estrategias para establecer un orden de selección entre las tareas_Y y tareas_O pendientes cuando se atiende una petición de un **procesador del mismo grupo**:

- **Estrategia 1:** Asignar las tareas más antiguas sin atender al tipo al que pertenecen.
- **Estrategia 2:** Dar prioridad a la asignación de las tareas_O.
- **Estrategia 3:** Dar prioridad a la asignación de las tareas_Y.

Las pruebas se han realizado con unos programas de prueba (*benchmarks* sintéticos que presentan paralelismo de ambos tipos con grano grueso. El siguiente programa (sintético 1) presenta paralelismo Y_bajo_O:

```
:- check.
check :- times1(X),p(X),p(X),p(X).
check :- times2(X),p(X),p(X),P(X).
check :- times3(X),p(X),p(X),P(X).
times1(2000).
times2(1000).
times3(500).
p(0).
p(X) :- X > 0, X1 is X - 1, p(X1).
```

Hay paralelismo_O en el procedimiento *check*. El paralelismo_Y aparece en el cuerpo de la cláusulas de este predicado. El siguiente programa (sintético 2) presenta parallelism O_bajo_Y:

```
:- check(X).
check([Xs,Ys,Zs]) :- times(X), times(Y), times(Z), p(X,Xs), p(Y,Ys), p(Z,Zs).
p(X,Xs) :- p1(X,Xs).
p(X,Xs) :- p2(X,Xs).
p1(0,a).
p1(X,Xs) :- X > 0, X1 is X-1, p1(X1,Xs).
p2(0,b).
p2(X,Xs) :- X > 0, X1 is X-1, p2(X1,Xs).
times(1000).
```

Los resultados obtenidos siguiendo cada una de las estrategias consideradas aparecen en la tabla 4.7. Cuando el número de procesadores del sistema es suficiente para asignar todas las tareas pendientes es indiferente el orden de explotación (sintético 1). Sin embargo, cuando no es así (sintético 2 con 8 procesadores), y lo que interesa es obtener todas las soluciones, se obtienen mejores resultados dando prioridad a la explotación del paralelismo_O, ya que se reparte trabajo de mayor granularidad. Asignando las tareas_Y en primer lugar se consigue obtener en menor tiempo las primeras soluciones, pero se retrasa la obtención de las últimas.

	8 proc.			*16 proc.*		
programa	estr. 1	estr. 2	estr. 3	estr. 1	estr. 2	estr. 3
sintético 1	4.5	4.5	4.5	4.4	4.4	4.4
sintético 2	2.3	2.7	2.3	3.4	3.4	3.4

Tabla 4.7: Mejora del rendimiento obtenida siguiendo distintos ordenes de explotación de los tipos de tareas

Cuando el controlador considera la asignación de una tarea a un **procesador perteneciente a un grupo distinto** al que posee tareas pendientes, selecciona una tarea_O. Las tareas_Y se ejecutan en el mismo grupo de procesadores básicos en el que se produjeron, ya que en este tipo de paralelismo es más importante la localidad pués el tráfico de mensajes entre los procesadores que comparten un trabajo es mayor (envío de respuesta al padre y backtracking).

Selección de una estrategia de asignación de tareas

El controlador puede seguir distintos criterios para realizar una asignación de trabajo cuando existe más de una tarea pendiente y más de una petición:

- **Proximidad topológica**
  Se asigna el trabajo pendiente al procesador desocupado que tiene asociado el mismo controlador y que está físicamente más próximo al donante. Sólo si no hay procesadores desocupados entre los correspondientes al mismo controlador se cede el trabajo a otro grupo.

- **Balance de carga**
  Se selecciona en primer lugar los trabajos pertenecientes al procesador básico con el mayor número de trabajos pendientes. Para ello se requiere conocer la carga de trabajo pendiente de cada procesador.

- **Antiguedad de los trabajos**
  Se seleccionan los trabajos más antiguos.

- **Granularidad**
  Se selecciona el trabajo de mayor grano entre los pendientes. Para seguir este criterio se requiere informar al controlador de la granularidad de cada trabajo pendiente.

- **Ejecuciones comunes**
  Al explotar el paralelismo_O en PDP se aprovecha la parte coincidente de la computación anterior y la futura, evitando su repetición. Por tanto se seleccionan las asignaciones de trabajo entre proceadores que lo han compartido previamente.

El orden de aplicación de estos criterios determina las distintas políticas de planificación. En PDP han sido investigadas las prioridades de aplicación de aquellos criterios que más pueden afectar al rendimiento del sistema. Teniendo en cuenta que todos los trabajos que han sido notificados al controlador tienen la sufienciente granularidad para que convenga su ejecución remota, nos centramos en los restantes criterios clasificando las estrategias consideradas para elegir el procesador donante que cede un trabajo de la siguiente forma:

- **Estrategia A: Por proximidad topológica**
  Se elige el procesador donante más cercano físicamente al procesador desocupado.

- **Estrategia B: Por balande de carga**
  Se elige el procesador donante con mayor carga de trabajo, aunque se mantiene la prioridad de cesión de trabajo entre procesadores del mismo grupo.

- **Estrategia C: Por Antiguedad**
  Se elige el procesador donante cuyo aviso se recibió antes.

La tabla 4.8 muestra la mejora del rendimento utilizando cada una de las estrategias en un sistema con 8 y 16 procesaores. Los resultados muestran que la mejor estrategia es la A, que elige el procesador donante más cercano al procesador desocupado que recibe el trabajo. Esta estrategia optimiza el tráfico de mensajes en la red y aumenta las probabilidades de que procesadores que han compartido trabajos previamente lo hagan de nuevo, lo que optimiza la explotación del paralelismo_O que evita la recomputación de la partes del entorno comunes entre el procesador donante y el receptor. La peor estrategia es la B, ya que es la que requiere un analisis más complejo y el envío de mayor cantidad de información para conocer la carga de trabajo de cada procesador. La estrategia C proporciona resultados mejores que la B ya que no necesita realizar ningún analisis. Las ventajas de la estrategia A sobre las otras se acentuan al aumentar el número de procesadores del sistema.

	8 proc.			16 proc.		
programa	estr. A	estr. B	estr. C	estr. A	estr. B	estr. C
query	2.9	2.8	2.8	3.4	3.2	3.3
zebra	3.7	3.5	3.6	3.9	3.5	3.6
mm	3.4	3.3	3.4	4.5	4.3	4.3
queen(8)	7.3	7.1	7.3	12.9	12.9	12.6
queen(9)	7.7	7.5	7.6	14.2	13.5	13.8
queen(10)	7.9	7.6	7.7	14.7	14.5	14.5

Tabla 4.8: Mejora del rendimiento utilizando distintas estrategias de planificación

4.4.5 Esquema general de planificación

Estableciendo un orden de aplicación de los criterios presentados se obtiene el esquema de selección de trabajos de PDP (Figura 4.8). Se selecciona el trabajo que correspondende al procesador más próximo al que hace la petición procedente de la explotación del paralelismo_O si lo hay. En otro caso se selecciona el más próximo procedente de la explotación del paralelismo_Y, siempre que se cumpla la restricción de la precedencia.

La Figura 4.9 describe el algoritmo de elección del procesador al que asignar un trabajo pendiente. Se selecciona entre los procesadores desocu-

```
procedure seleccionar_trabajo(peticion:integer; tabla_trabajos: t_traba;
                  num_tra: integer; VAR trabajo: integer);
var
    nt: integer; /* contador de trabajos comprobadas */
begin
    /* Se selecciona el trabajo de paralelismo_O mas proximo */
       trabajo := selec_proximo_parO(trabajo, tabla_tra, num_tra);
       if (trabajo = 0) then
          trabajo := selec_proximo_parY(trabajo, tabla_tra, num_tra);
end
```

Figura 4.8: Algoritmo de selección del procesador con trabajo pendiente que atenderá una petición de trabajo.

pados el que se encuentra físicamente más próximo al procesador que posee el trabajo. Si el trabajo procede de la explotación del paralelismo_Y se comprueba que se cumpla la condición de precedencia. Si no se cumple se pasa a considerar otro de los procesadores desocupados.

```
procedure seleccionar_procesador(trabajo: integer; tabla_pet: t_pet;
                num_pet: integer; VAR peticion: integer);
var
    np: integer; /* contador de peticiones comprobadas */
begin
    seleccionado := false;
    np := 0;
    while not seleccionado and (np < num_pet) do
    begin
        /* Se busca el procesador desocupado mas proximo */
        peticion := selec_proximo(trabajo, np, tabla_pet);
        /* Si es de paralelismo_Y se comprueba la precedencia */
        if tipo(tabla_trabajo[trabajo] <> Y or precedencia(trabajo,peticion) then
            seleccionado := true
        else
            np := np + 1;
    end
    if not seleccionado then peticion := 0;
end
```

Figura 4.9: Algorítmo de selección del procesador al que se asigna un trabajo pendiente.

5

Implementación de PDP sobre una Red de Transputers

5.1 Introducción

PDP ha sido desarrollado para ser implementado en una arquitectura multiprocesadora con la memoria distribuida, en la que la comunicación entre procesadores se realiza únicamente por paso de mensajes. Aunque puede ser implementado sobre cualquier arquitectura de este tipo, actualmente lo ha sido sobre un red de transputers, un dispositivo programable en un único *chip* con un alto rendimiento, que ha sido desarrolado por la compañia *Inmos*, para facilitar el diseño de los sistemas multiprocesadores debilmente acoplados. Consta de un procesador de 32-bit basado en ideas RISC con una memoria RAM interna estática, un interface de memoria configurable para acoplar memoria externa (lo que simplifica el diseño de sistemas de transputers), cuatro enlaces de comunicación de alta velocidad (20 Mbit/s en el transputer T800) para la conexión con otros transputers y un interface opcional para aplicaciones específicas como una unidad de control de disco o un coprocesador en punto flotante. El transputer facilita el tratamiento de la concurrencia, ya que a diferencia de otros procesadores en los que el tratamiento del proceso multitarea se realiza por software, mediante el sistema operativo, en el trasputer este tratamiento está contenido en el microcódigo del procesador, por lo que proporciona un rápido cambio de contexto. Existen diferentes entornos de programación para el desarrollo de software para transputers. Occam constituye el más bajo nivel en que los transputers pueden ser programados, pudiendo verse el transputer como

una máquina virtual para la ejecución de programas en Occam. Aunque es posible programar los transputers en una mezcla de lenguajes, de forma que secciones de código fuente escritas en diferentes lenguajes formen parte de la implementación, en la práctica presenta diversos problemas [44]. Por estas razones PDP ha sido desarrollado en ANSI C paralelo de Inmos [79] , que es compatible con el diseño de Kernighan y Ritchie [46] y soporta el paralelismo mediante librerias de procedimiento de comunicaciones y tratamiento del procesamiento multitarea.

Cada nodo de la red implementa un procesador básico o un controlador en los que se ha separado las funciones de procesamiento y comunicaciones para prevenir bloqueos. Se ha estudiado la topología de la red más apropiada para PDP, optando por una red toroidal. Se presenta también el procedimiento de encaminamiento de mensajes necesario para la topología elegida. Finalmente se presentan los detalles del desarrollo de PDP.

5.2 Asignación de Procesos a Procesadores

La asignación de tareas a los procesadores de una red puede ser estática o dinámica. La primera sólo es posible si se conocen *a priori* todas las tareas paralelas. Debido a la limitada conectividad de los transputers (4 enlaces), la asignación de las tareas correspondientes a más de un procesador básico a un mismo trasputer obligaría a multiplexar por software las comunicaciones en los enlaces de comunicación introduciendo un alto recargo en el tiempo de comunicación. Los procesadores básicos que procesan trabajos procedentes de la explotación del paralelismo_O y del O_bajo_Y, pueden ocupar todo el tiempo de CPU del procesador ya que no se producen tiempos de espera. Por otra parte, los procesadores básicos encargados de estos trabajos después de terminar la computación pueden realizar cualquier nuevo trabajo del sistema. Por tanto se ha optado por asignar a cada procesador los procesos correspondientes a un único procesador básico, evitando así la multiplexación de los enlaces. Un procesador básico que explota paralelismo_Y o que procesa un trabajo procedente de la explotación del paralelismo_Y, pueden producirse esperas de respuestas. En estos casos, para evitar que un procesador se encuentre parado mientras queda trabajo pendiente en el sistema, se ha optado por permitir la creación de un nuevo procesador básico en el procesador. Este nuevo procesador básico únicamente se ocupará de trabajos derivados de la explotación del paralelismo_Y que ha originado la

espera. De esta forma al terminar el tiempo de espera los nuevos procesadores básicos pueden desaparecer evitando así la multiplexación de los canales. Por tanto, el esquema de asignación de procesos a procesadores queda de la siguiente forma:

- Inicialmente se hace una asignación estática de los procesos correspondientes a los procesadores básicos y a los controladores a los transputers.
- Esta asignación se mantiene siempre que ese explota paralelismo_O u O_bajo_Y
- Si al explotar paralelismo_Y un procesador básico queda en estado de espera, crea un nuevo procesador básico en el mismo transputer que puede encargarse de trabajos derivados del paralelismo_Y explotado. Los nuevos procesadores básicos del transputer desaparecen al terminar el tiempo de espera

Este esquema se realiza con la intervención del controlador, que registra la procedencia de los trabajos en la explotación el paralelismo_Y, pudiendo de esta forma asignar a los procesadores básicos creados dinámicamente los trabajos apropiados.

5.3 Procesos de un procesador básico

Los enlaces de comunicación entre transputers realizan una comunicación en serie. Estos enlaces son síncronos (Figura 5.1), por lo que no se necesita almacenamiento. Un proceso que va a enviar un mensaje espera a que el proceso receptor esté preparado para recibirlos y vice versa (*rendez-vous*). Este sincronismo puede dar lugar a problemas de bloqueo en las comunicaciones, ya que si dos porcesos estan intentando enviarse un mensaje mutua y simultaneamente, ninguno de los dos será atendido por el otro y se bloquearán, pudiendo darse esta situación entre más procesadores (Figura 5.2).

Para evitar posibles bloqueos, en cada procesador se han separado las funciones de comunicación y de procesamiento, existiendo tres procesos encargados de la entrada, la salida y el procesamiento respectivamente según se representa en la Figura 5.3. De esta forma aunque el proceso de salida se quede en espera de atención por parte de transputer destino, el proceso de

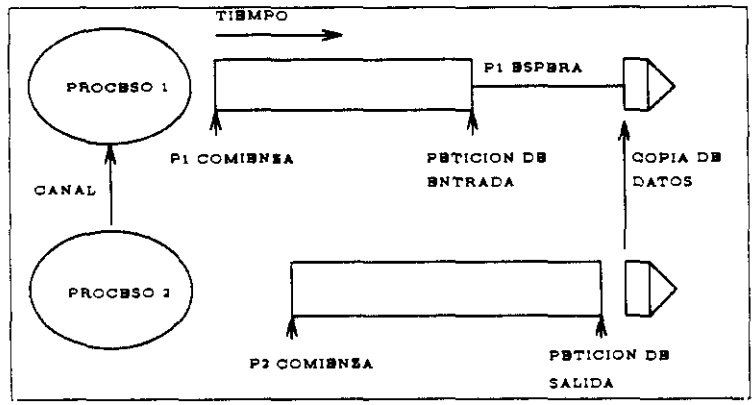


Figura 5.1: Sincronización de procesos

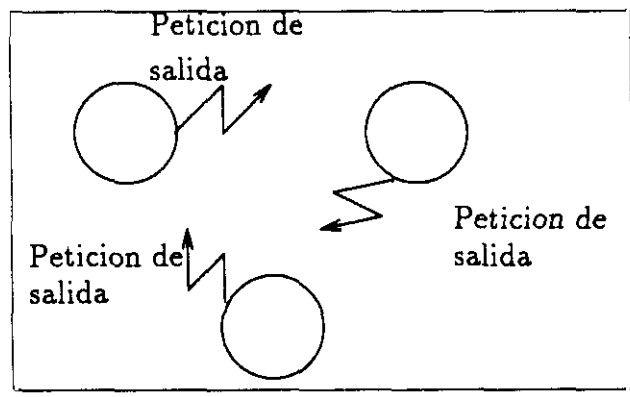


Figura 5.2: Situación de bloqueo

entrada puede seguir recibiendo mensajes que almacena, impidiendo así que otros transputers queden también a la espera.

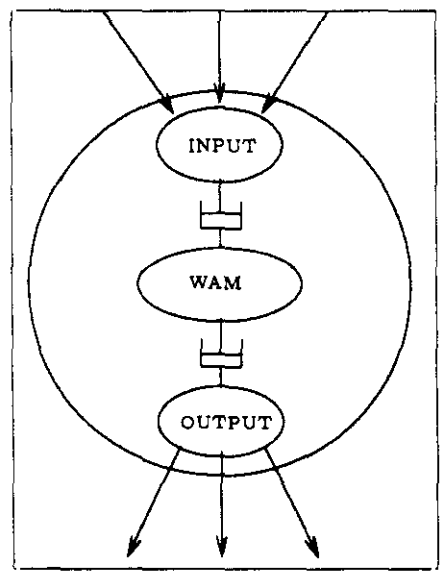


Figura 5.3: *Esquema de procesos en un procesado básico*

El proceso de entrada está suspendido hasta que llega un mensaje por cualquiera de los canales de entrada del procesador. Los mensajes se almacenan en una zona global de memoria (mensaje_ent) cuyo acceso está controlado por un semáforo (sema_ent), y desde la que son tomados para su tratamiento.

```
{{{  InputProcess }}}
static void InputProcess (Process *InputP, Channel *Input[])
{
  signal(SIGUSR1, SIG_IGN);
  while (1) {
    Index = ProcAltList(Input);
    SemWait(sema_ent);
    recibir_mensaje(Input[Index], mensaje_ent);
    SemSignal(sema_ent); }
}
```

En primer lugar el proceso se protege de la interrupción software SIGUSR1 mediante la instrucción *signal*, de manera que no tenga efecto sobre

él. La instrucción *ProcAltList* deja el proceso suspendido hasta que se recibe un mensaje por cualquiera de los canales de *Input*. Cuando esto ocurre devuelve a *Index* el número del canal por el que se ha producido la entrada. Con la instrucción *SemWait* se solicita el semáforo de acceso a la zona de memoria utilizada para la entrada *sema_ent*. Después de recibir el mensaje se libera el semáforo (instrucción *SemSignal*).

El proceso dedicado a la interpretación de código WAM está continuamente ejecutando instrucciones hasta que es avisado de la llegada de mensajes.

```
{{{  WamProcess  }}}
static void WamProcess (Process *WamP)
{
  signal(SIGUSR1, SIG_IGN);

  while(1) {
    if hay_mensaje
      trata_mensaje();

    if(estado == ACTIVO)
      ejecuta_instruccion(opcode);
  }
}
```

El proceso se protege de la interrupción software SIGUSR1 y a continuación entra en un bucle de tratamiento de mensajes e instrucciones. Dentro de este bucle puede necesitar enviar un mensaje, por ejemplo para avisar de que se ha producido un nuevo trabajo. Para avisar al proceso de salida de que tiene un mensaje para enviar se activa la interrupción software SIGUSR1 mediante la instrucción *raise*.

El proceso de salida esta inactivo hasta que es despertado mediante una interrupción software (signal) cuando se necesita enviar un mensaje.

```
{{{  OutputProcess  }}}
static void OutputProcess (Process *OutputP, Channel *Output[])
{
  signal(SIGUSR1, sig_cap);
  while(1) {
    ProcReschedule();
    SemWait(sema_sal);
    envia_mensaje(mensaje_sal);
```

```
    SemSignal(sema_sal);}
}
sig_cap()
{
  signal(SIGUSR1, SIG_IGN);
}
```

El proceso se prepara para detectar la interrupción software SIGUSR1 cuando se produzca (instrucción *signal*). A continuación mediante la instrucción *ProcReschedule* el proceso se pone al final de la cola de planificación de procesos, de manera que no se activará mientras el resto de procesos esté activo o no llege la señal SIGUSR1. Cuando otro proceso desea enviar un mensaje activa la señal que despierta al proceso de salida. Este solicita el semáforo de acceso a la zona de memoria utilizada para la salida, envia el mensaje y libera el semáforo.

5.4 Protocolo de mensajes

En los sistemas distribuidos es necesario especificar y validar el diseño del protocolo de comunicaciones. En PDP se ha utilizado un sistema de transiciones para modelizar el protocolo. Un sistema de transiciones tiene un conjunto de estados y un conjunto de transiciones entre dichos estados. Emplearemos una representación similar a la utilizadada por Zafiropulo *et al.* [77]. Los tipos de mensajes que intercambian entre el controlador y los procesadores básicos de PDP son los siguientes:

- **CODIGO**
  Es un mensaje enviado por el controlador a cada procesador básico conteniendo el código de programa y la tabla de símbolos

- **PETICION**
  Es una mensaje que envian los procesadores básicos sin trabajo al controlador pidiendo trabajo.

- **TRABAJO_PENDIENTE**
  Es enviado al controlador por los procesadores básicos con trabajo pendiente.

- **INTERCAMBIO**
  Es una orden de intercambio de trabajo que envía el controlador a

un procesador básico con trabajo pendiente, indicandole el procesador desocupado al que debe enviarselo.

- **CONFIRMACION**
  Es enviado al controlador por un procesador básico que ha recibido trabajo de otro.

- **NEGACION**
  Es enviado al controlador por un procesador básico que ha recibido un mensaje de INTERCAMBIO que no puede satisfacer por haber dejado de tener trabajo pendiente.

- **BORRAR**
  Los procesadores básicos que dejan de tener trabajo pendiente envian este mensaje al controlador para informarle de ello.

- **RESPUESTA**
  Cuando un procesador básico con una taarea de tipo nodo_O termina la ejecución que se le habia asignado, envia la respuesta de éxito o fallo al controlador mediante este mensaje, para ser presentado al usuario.

Los posibles estados del controlador son:

- **Libre**
  Es el estado inicial y significa que el controlador no tiene pendientes ni peticiones ni trabajos que atender.

- **Esp_petición**
  En este estado el controlador que ha sido informado de la existencia de trabajo pendiente en el sistema, espera la petición de un procesador desocupado para dar la orden de intercambio de trabajo.

- **Esp_trabajo**
  El controlador ha sido informado de la existencia de procesadores libres en el sistema y espera un aviso de trabajo pendiente para dar la orden de intercambio.

- **Tratamiento_peticiones**
  Cuando hay trabajo pendiente en el sistema y procesadores libres en el sistema, el controlador hace un analisis del intercambio de trabajo más apropiado según los criterios del capítulo 4.

- **Esp_confirmación**
  Después de dar una orden de intercambio de trabajo, el controlador espera la confirmación de que ese intercambio se ha producido efectivamente.

- **Esp_negación**
  Si una orden de intercambio de trabajo del controlador se ha producido al mismo tiempo que un aviso de borrar ese trabajo por parte del procesador que lo tenía pendiente, se recibirá una negación a dicha orden de intercambio.

El mensaje **CODIGO** conteniendo el código del programa es enviado por el controlador a los procesadores básicos inicialmente y no ha sido representado en el protocolo de comunicaciones para simplificar el esquema. La Figura 5.4 describe el protocolo de mensajes entre el controlador y los procesadores básicos. Los mensajes con que se etiquetan las transiciones van acompañados de un signo que indica si son de entrada (+) o salida (−).

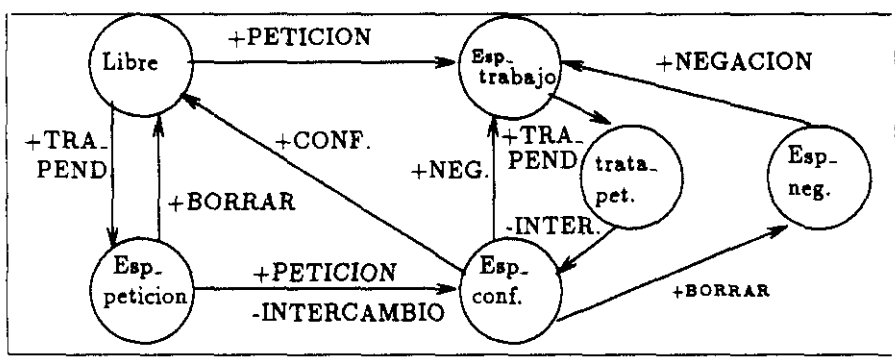


Figura 5.4: Protocolo de mensajes entre el controlador y los procesadores básicos

El controlador comienza en estado **Libre**. Si recibe una **PETICION** pasa a estado de espera de un aviso de trabajo pendiente **Esp_trabajo**. Cuando recibe un mensaje de **TRABAJO** pasa a **Tratamiento_peticiones**, estado en el que analiza que petición de las recibidas es conveniente atender en primer lugar, dando entonces una orden de **INTERCAMBIO** y pasando a estado de espera de la confirmación de que se ha producido el intercambio (**Esp_confirmación**). Cuando recibe el mensaje **CONFIRMACION** del

intercambio de trabajo, pasa a estado **Libre**. Si en lugar de una confirmación recibe un aviso del temporizador, indicando que ya debería haberse recibido la confirmación, tras anotar de nuevo la petición correspondiente al intercambio que no ha sido confirmado, vuelve al estado de **Esp_trabajo**. Si en lugar de la confirmación se recibe un mensaje de **BORRAR** el trabajo que se había ordenado intercambiar, se pasa al estado **Esp_negación** en el que se espera el mensaje **NEGACION** que envía el procesador que tenia el trabajo. Cuando llega la negación se vuelve al estado **Esp_trabajo**.
Los estados en que puede encontrarse un procesador básico son:

- **Libre**
  Se trata del estado inicial.
- **Esp_trabajo**
  El procesador esta desocupado.
- **Activo**
  El procesador está computando el trabajo recibido.
- **Activo_con_trabajo_pendiente**
  El procesador está computando el trabajo recibido, en el que ha encontrado paralelismo y por tanto puede ceder trabajo a otros procesadores.
- **Esp_respuesta**
  El procesador está esperando la respuesta a un trabajo que envió a otro procesador.

Los mensajes intercambiados entre procesadores básicos son:

- **TRABAJO**
  Es un mensaje que contiene trabajo pendiente.
- **RESPUESTA**
  Se trata de la respuesta que un procesador que recibió una tarea_Y, envía al procesador que le envió dicha tarea.
- **REDO**
  Es enviado por un procesador que ha enviado una tarea_Y al procesador al que se la envió, para pedirle una nueva respuesta a dicha tarea.

- **KILL**
  Es una orden enviada por un procesador que ha enviado trabajo al que se lo envió, para que desheche el trabajo recibido.

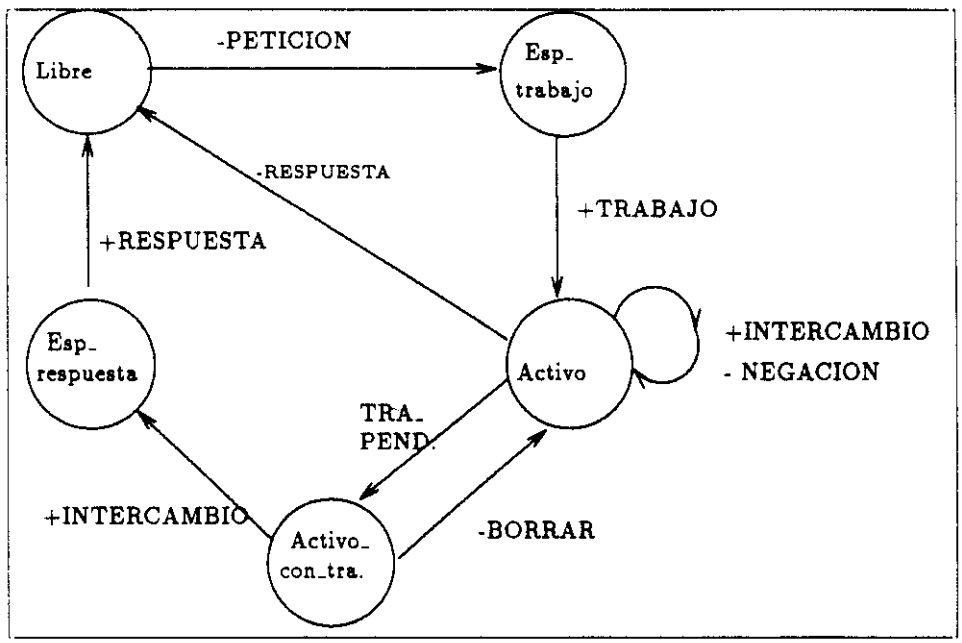


Figura 5.5: Protocolo de mensajes entre procesadores básicos

El protocolo de comunicaciones entre procesadores básicos se describe en el diagrama de la Figura 5.5. Un procesador básico se encuentra inicialmente en estado **Libre**. Tras enviar un mensaje de **PETICION** al controlador pasa a estar en estado de espera de trabajo **Esp_tra**, pasando a estar **Activo** cuando recibe un mensaje de **TRABAJO**. Si en este estado recibe ordenes de **INTERCAMBIO** del controlador, contesta con un mensaje de **NEGACION**. Si durante la ejecución del trabajo recibido aparece paralelismo y se producen trabajos pendientes, el procesador pasa a estado **Activo_con_trabajo_pendiente**, enviando un mensaje de **TRABAJO_PENDIENTE** al controlador. Si en este estado recibe una orden de intercambio del controlador, envía el trabajo pendiente y pasa a estado de **Esp_resp**, del que vuelve al recibir el mensaje de **RESPUESTA**. Si estando **Activo_con_trabajo_pendiente**, ejecuta él mismo dicho trabajo

pendiente, pasa a estado **Activo** enviando un mensaje de **BORRAR** al controlador. La verificación de este protocolo se presenta en el apendice B.

5.4.1 Formato de mensajes

La cabecera de un mensaje consta de los siguientes campos:

- tipo
- origen
- destino
- longitud

Esta cabecera es común a todos los mensajes del sistema y facilita el analisis y el encaminamiento de los mensajes. El resto del mensaje depende del tipo de mensaje de que se trate. Algunos como el de *PETICION* consisten únicamente en la cabecera. Los mensajes de *RESPUESTA* contienen información sobre el trabajo al que corresponden y sobre si la respuesta es de éxito o de fallo. Los mensajes de *TRABAJO_PENDIENTE* contienen información sobre la carga de trabajo.

5.5 Elección de la Red de Interconexión

La red de interconexión se elige de acuerdo a la aplicación a la que se destina, optimizando los parametros que más interesen en cada caso. Un factor fundamental en la elección de la red es la *distribución de encaminamiento* de mensajes, que se define como la probabilidad de que diferentes nodos intercambien mensajes. Una distribución de encaminamiento de mensajes es *uniforme* si la probabilidad de que un nodo i envie un mensaje a un nodo j es la misma para todo i y j, $i \neq j$. La distribución de encaminamiento de mensaje no es uniforme en general. Un modelo que en el que se engloba el comportamiento de PDP es el que considera que las tareas que se comunican frecuentemente se colocan próximas en la red. Una abstracción de esta idea coloca cada nodo en el centro de una esfera de localidad. Un nodo envía mensajes a otros nodos de su esfera de localidad con una probabilidad alta ϕ, mientra que envía mensajes a nodos fuera de la esfera con una baja probabilidad $1 - \phi$. En PDP los elementos de proceso asignados a un

mismo controlador se comunican con una probabilidad significativamente mayor que con el resto. Por tanto podemos considerar que los nodos asociados a un controlador se comunican con una distribución de probabilidad uniforme, mientras que en la red completa se distinguen esferas de localidad correspondientes a cada controlador.

Entre las redes de interconexión propuestas [63] hay una serie de ellas que proporcionan un punto de referencia sobre las posibles configuraciones de una red o tienen características especialmente atractivas:

- **en bus**
- **completamente conectada**
- **anillo**
- **hipercubo de bus expandido** Es un enrejado de dimensión D.
- **hipercubo binario** Se obtiene eliminando $D-2$ conexiones de cada nodo de la red anterior.
- **toro** Es como la anterior pero el bus que conecta cada grupo de w nodos se reemplaze por un anillo.
- **arbol**
- **N-cubo R-ario** Es una generalización del hipercubo binario, donde N es el número de niveles y R^N es el número de nodos en cada nivel. La red contiene NR^N nodos, cada uno de ellos conectados a $2R$ nodos.

Para realizar la elección de la red se han considerado una serie de factores de los cuales algunos han sido impuestos por el tipo de procesadores (transputers) que la componen, y otros son requisitos necesario para mejorar el rendimiento de PDP:

- **Limite en el número de enlaces de cada nodo**
  Puesto que un transputer solo dispone de 4 enlaces de conexión, quedan eliminadas las topologías que requieren un número superior como la de una red completamente conectada.
- **Pequeña distancia media entre nodos**
  La distancia media es el número de saltos de un mensaje "típico" para

llegar a su destino y constituye un indicador del retardo medio de los mensajes. Su valor depende de la distribución de encaminamiento de mensajes. Para una distribución uniforme, el hipercubo de bus expandido presenta la menor distancia media en redes de más de 20 nodos.

- **Incrementos de expansión de la red arbitrarios**
  El incremento del tamaño de la red no siempre puede ser arbitrario, existiendo muchos sistemas que solo pueden expandirse en incrementos del tamaño actual de la red. Por ejemplo el tamaño de un hipercubo solo puede incrementarse doblando el número de nodos. Son preferibles los incrementos pequeños por que permiten una mayor flexibilidad en el diseño. Mientras algunas redes, como los N-cubos R-arios se expanden muy rápidamente, otras como el toro tienen incrementos muy pequeños para dimensiones limitadas (D = 2,3,4,...).

- **Escalabilidad de la Red**
  Idealmente, debe ser posible crear redes multicomputadoras mayores y más potentes simplemente añadiendo más nodos a la red. Sin embargo, la escalabilidad puede estar restringida por diversos factores, pudiendo aparecer cuellos de botella al incrementar el tamaño, como puede ocurrir en los nodos cercanos a la raíz de los árboles. El rendimiento de estas redes tiene un límite superior dado por una constante como ocurre con la red en bus, anillo o arbol. Puesto que la mejora del rendimiento mediante la expansión de la red es importante, estas redes no se consideran.

- **Alto rango de paso de mensajes**
  Cada mensaje que se envía atraviesa una serie de enlaces y nodos intermedios y produce una computación en el nodo destino. El conjunto de enlaces cruzados y la computación en el nodo destino constituyen una *visita* a ese enlace o elemento de proceso. Si se consideran todos los pares de nodos fuente-destino y la probabilidad de que intercambien mensajes, puede calcularse el número de visitas de un *mensaje medio* a cada enlace y nodo. Dividiendo por el número de enlaces de la red se obtiene el *Porcentaje de visitas*. Esta cantidad puede verse como una medida de la intensidad de mensajes que soporta un enlace. Si es próximo a 1, entonces la mayoría de los mensajes deben cruzar cada enlace en algún momento del camino hacia su destino. Las redes en

hipercubo y toroidal soportan con diferencia el mayor rango de paso de mensajes sin llegar a la saturación.

Las consideraciones anteriores indican que la configuraciones en hipercubo y toroidal presentan las características más apropiadas para la implementación de PDP. En la implementación actual se ha optado por una red que permite incrementos de expansión pequeños ya que se ha realizado sobre un número pequeño de procesadores (Figura 5.6). Se ha adoptado un red toroidal de dimensión 2 que agrupa a 16 transputers T800. La distribución de mensajes presenta esferas de localidad centradas en los transputer controladores, existiendo una distribución uniforme dentro de cada una de estas zonas.

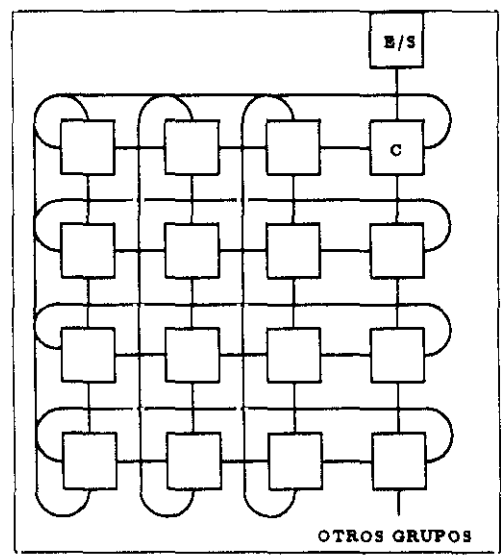


Figura 5.6: Topologia de la red en PDP

El controlador se coloca en un transputer del extremo de la red para poder conectarlo al procesador encargado de las funciones de entrada/salida, del que recibe el código de programa y al que envia los resultados que son presentados al usuario.

5.5.1 Análisis de la Configuración Toroidal

Una red toroidal de dimensión D, conecta cada uno de sus w^D nodos en un anillo de tamaño w en cada una de las D dimensiones ortogonales. No solo el hipercubo binario es una caso especial del hipecubo de bus espandido, sino que también es un caso especial del toro cuando $w = 2$ y el anillo de dos nodos se reemplaza por un único enlace. Puesto que cada w^D nodos están conectados a D anillos, existen Dw^D enlaces. La distancia media entre nodos en una red toroidal con distribución de mensajes uniforme viene dada por:

$$DM = \begin{cases} \dfrac{Dw^{D+1}}{4(w^D-1)} & \text{w par} \\ \dfrac{Dw^{D+1}(w^2-1)}{4(w^D-1)} & \text{w impar} \end{cases}$$

que en la implementación actual de PDP (D = 2, w = 4) dá una distancia media de 2, es decir, los mensajes atraviesan una media de 2 enlaces.

La simetria de la red nos permite obtener inmediatamente a partir de este dato el porcentaje de visitas normalizando con el número de enlaces:

$$V = \frac{DM}{Dw^D} = \begin{cases} \dfrac{w}{4(w^D-1)} & \text{w par} \\ \dfrac{w^2-1}{4(w^D-1)} & \text{w impar} \end{cases}$$

que en la implementación actual de PDP dá un porcentaje de visitas de 1/15, es decir, de cada 15 mensajes que se intercambian en la red, uno atraviesa el enlace considerado.

Finalmente el toro con su estructura de enrejado, tiene un incremento de expansión de

$$(w+1)^D - w^D$$

es decir, para pasar de un toro de dimensión 2 con 3 nodos por anillos a uno con 4 nodos por anillo se necesitan únicamente 7 nodos adicionales.

5.5.2 Influencia de la Topologia de la red en la Granularidad

El porcentaje de visitas puede usarse para determinar los granos de computación adecuados, dada la velocidad relativa entre los procesadores y los

enlaces de comunicación. Del desarrollo realizado en [56] resulta que si K es el número de nodos de la red, el porcentanje entre el tiempo de computación y el tiempo de comunicación para un mensaje debe ser al menos K veces el porcentaje de visitas máximo de los enlaces de la red si no se quiere limitar el porcentaje de computación por el retardo de las comunicaciones. Por lo tanto en la implementación actual de PDP, en que $K = 16$ y el porcentaje de visitas máximo es de $1/15$ (coincide con el medio por la simetría de la red), el tiempo de computaciones realizadas en paralelo debe ser al menos del orden del tiempo necesario en realizar la comunicación que dá origen a la computación.

5.6 Encaminamiento

Toda red de comunicaciones necesita un algoritmo de encaminamiento para construir el camino entre los nodos que intercambian mensajes. Por ser regular la red de PDP, el encaminamiento se puede realizar en cada nodo mediante un algoritmo fijo basado en las direcciones local y destino. El procedimiento encargado del encaminamiento dirige el mensaje en la dirección que reduce la diferencia entre las coordenadas x o y del nodo actual y el destino:

```
router(dir,salida)
char dir; char *salida;
{
  char filp;   char colp;  /* fila y columna propia */
  char fild;   char cold;  /* fila y columna destino */
  char difcol; char diffil;/* distancia entre columnas y filas */

  filp = nwam / NW;  colp = nwam % NW;
  fild = dir / NW;  cold = dir % NW;

  /* se comprueba si estan en la misma fila */
  if (filp == fild) {
    difcol = cold - colp;
    if (difcol > 0) {
      if (difcol <= NW/2)
        *salida = CANAL_ESTE;
      else
        *salida = CANAL_OESTE;}
    else {
```

no. reinas	sin router	con router
4	269	301
5	2408	2624
6	3912	3931
7	17616	18093
8	73742	76376

Tabla 5.1: *Recargo introducido por el encaminamiento*

```
      if ((NW + difcol) <= NW/2)
        *salida = CANAL_ESTE;
      else
        *salida = CANAL_OESTE;}}
  else {
    diffil = fild - filp;
    if (diffil > 0) {
      if ((diffil <= NW/2) || (nwam == WAM_DESCONEC) || (nwam == N_WAM))
        *salida = CANAL_NORTE;
      else
        *salida = CANAL_SUR; }
    else {
      if (((NW + diffil) <= NW/2) || (nwam == WAM_DESCONEC) || (nwam == N_WAM))
        *salida = CANAL_NORTE;
      else
        *salida = CANAL_SUR;}}
}
```

La tabla 5.1 muestra el recargo que supone la introducción del encaminamiento en una red formada por 4 nodos, comparando las medidas obtenidas cuando cada nodo estaba conectado con cada uno de los restantes y por tanto no era necesario encaminar los mensajes, y cuando los nodos estan conectados en una red toroidal y utilizan el procedimiento de encaminamiento. El recargo que se introduce es menor del 0.1

Algunas de las limitaciones de los transputers son superadas por el nuevo transputer T9000, que proporciona facilidades para el encaminamiento de mensajes y la multiplexación y demultiplexación de los paquetes de datos en los enlaces. El T9000 se complementa con un periférico de tratamiento de comunicaciones, el C104 que es un *chip* conmutador de encaminamiento. El C104 conecta 32 enlaces entre sí mediante conexiones de latencia de sub-

μs. Un único C104 permite conectar entre sí directamente a 32 transputers T9000. Los C104s también pueden conectarse entre sí para construir mayores redes de conmutación que permitan conectar un número más alto de transputers. Esto permite sencillas y rápidas comunicaciones entre transputers T9000 que no están conectados directamente (Figura 5.7).

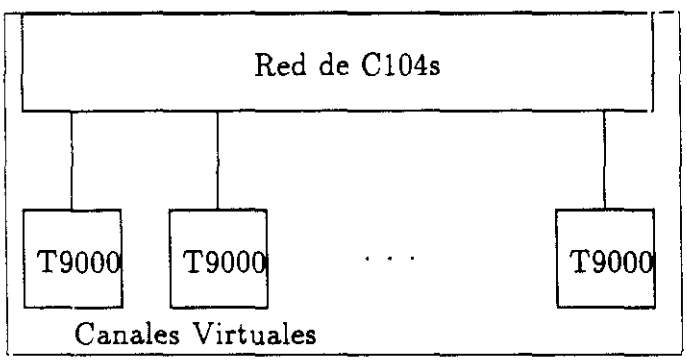


Figura 5.7: Sistema de encaminamiento de T9000

Los transputers anteriores al T9000 implementan los canales entre procesos del mismo procesador mediante zonas de memoria, y entre procesos de distintos procesadores mediante enlaces punto a punto. Como cada procesador dispone únicamente de 4 enlaces cuando se necesita un número mayor de canales se introducen procesos multiplexadores, que representan un alto recargo y reducen la utilización de los enlaces. El T9000, además de su CPU, incorpora un *procesador de canales virtuales* para multiplexar cualquier número de enlaces virtuales a cada enlace físico. De esta forma los programas escritos para una determinada topología de la red pueden ser portados a otra diferente. El procesador de canal virtual del T9000 también se utiliza para encaminar los mensajes a través del sistema de comunicaciones que conecta cualquier número de transputers T9000. De esta forma *el programador no necesita especificar la forma de encaminar los mensajes.* Los mensajes no pasan a través de transputers intermedios reduciendo el rendimiento de estos. El T9000 facilitará la realización de futuras implementaciones de PDP en una red con un alto número de transputers, facilitando la implementación de distintas topologias de la red.

5.6.1 Sistemas soporte utilizados en la implementación de PDP

Existe una familia de transputers compuesta por diferentes procesadores: de 16-bit (T212,T222), de 32-bit sin procesamiento de números en punto flotante (T414,T425), con procesamiento en punto flotante (T800,T801) y y el último miembro, el T9000. Existe también una placa de 32 por 32 conmutadores, la IMS C004, controlada por comandos sobre enlaces especiales para redes reconfigurables (Figura 5.8). La reconfigurabilidad es necesaria para ajustar la topología de la red de transputers a la aplicación. PDP ha sido implementado usando el T800 y se prevee realizar futuros desarrollos utilizando las capacidades de comunicación y rendimiento del T9000.

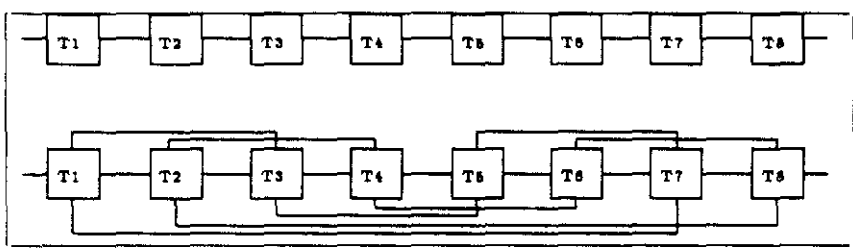


Figura 5.8: Red reconfigurable

Actualmente existen diferentes plataformas para el desarrollo de un sistema de transputers, de los que dos han sido utilizados para el desarrollo de PDP:

- **PC**
  Ha sido el más extendido durante algún tiempo por lo que es el mejor soportado por diversas compañias. El desarrollo inicial de PDP fue realizado en un PC con 5 transputers T800 sobre una placa IMS C004, utilizando el entorno de programación ICTOOLS de Inmos.

- **Supernodo (Parsys)**
  Es un sistema basado en transputers con el sistema operativo *IDRIS*, que ofrece un entorno de programación comodo y de alta flexibilidad en el uso de lenguajes de alto nivel. Es compatible con los entorno de desarrollo de Inmos.

La serie SN 1000 de Parsys presenta un arquitectura reconfigurable, es decir, la arquitectura puede modificarse para ajustarla al problema tratado. La arquitectura de estos sistemas está basada en la desarrollada para el Supernodo ESPRIT P1085. Los modulos de transputers se organizan en nodos que pueden ser conectados de forma muy flexible para configurar sistemas de hasta 64 nodos. Cada nodo consta de una serie de transputers que proporcionan los recursos computacionales y otra serie de transputers que soportan la gestión de entrada/salida y las funciones de control, es decir, la reconfiguración. En una configuración mínima hay 16 transputers dedicados a la computación (organizados en 2 placas de 8) y un transputer de control. Este transputer proporciona acceso al conmutador de programación de los enlaces por software que permite al usuario configurar la red en una topología específica. El bus de control es accesible por todos los transputers.

Los Supernodos de esta serie incorporan el sistema operativo *Idris*, que es de tipo UNIX. Se trata de un entorno multi-usuario, multi-tarea extendido con un conjunto de utilidades para la gestión de los transputers del sistema, permitiendo cargar un entorno de desarrollo Occam o C para ser ejecutados en un transputer seleccionado o en un grupo de ellos.

5.7 Depuración en PDP

El desarrollo y depuración de PDP se ha visto complicado por la elección hecha de la configuración de la red de comunicaciones que utiliza todos los enlaces de los transputers componentes para mejorar el rendimiento. Al no quedar ningún canal libre no ha sido posible la depuración interactiva manteniendo la configuración del sistema. Por este motivo, para realizar la depuración hemos situado todos los procesadores básico en mismo transputers conectado al procesador encargado de la entrada/salida, pudiendo de esta forma visualizar el estado en los puntos que interesaba. La depuración del modelo por copia para la explotación del paralelismo_O ha presentado especiales dificultades ya que el espacio de direcciones de memoria en este modelo ha de ser identico en todos los procesadores, y por tanto no ha sido posible aplicar la misma técnica de depuración. Para poder visualizar el estado que va a ser enviado y también el obtenido después de la transmisión, la máquina que detecta el paralelismo se inicializa después de enviar el estado al controlador y éste se lo devuelve. De esta forma se puede comprobar si el estado original y el reconstruido a partir del mensaje coinciden.

Los sistemas multiprocesador distribuidos no están provistos de un reloj único que pueda ser leido por todos los procesos. Esto supone un obstaculo para el desarrollo de herramientas de depuración y evaluación del rendimiento. En PDP se ha optado por realizar todas las medidas utilizando el reloj del controlador.

La implentación de PDP con transputers T9000 facilitará la depuración pués mediante los canales virtuales proporciona el canal adicional que es necesario como mínimo para realizar la depuración interactiva. Además, proporciona un mecanismo de tratamiento de errores que permite obtener información sobre los fallos ocurridos, disponiendo de un tipo especial de proceso denominado *protegido* que se ejecuta bajo el control de un proceso supervisor al que es devuelto el control en caso de error.

6

Resultados

6.1 Introducción

La ley de *Amdahl* predice una mejora del rendimiento limitada en los sistemas paralelos debido a que la velocidad del sistema está limitada por su parte secuencial. Independientemente del número de procesadores paralelos un problema nunca puede resolverse más rápidamente de lo que permita su parte secuencial. Sin embargo cuando el tamaño del problema se incrementa, la explotación del paralelismo puede permitir mantener el tiempo de ejecución constante utilizando más procesadores en la ejecución (ley de *Gustafson-Barsis*). Por tanto la medida del rendimiento en un sistema paralelo necesita parametros distintos de los tradicionalmente usados en los sistemas secuenciales (número de instrucciones o inferencias lógicas por unidad de tiempo) que permitan caracterizar los efectos de la explotación del paralelismo en cada tipo de problema. La medida de rendimiento paralelo más utilizada es la curva de *speedup*. Se obtiene dividiendo el tiempo necesario para obtener la solución al problema considerado utilizando un sólo procesador por el tiempo necesario utilizando N procesadores.

En este capítulo se evaluan las ideas presentandas en los anteriores. Para cada tipo de paralelismo se han investigado los siguientes paramétros:

- La **curva de speedup** y su relación con el paralelismo medio de los programas.
- **Recargo** que introduce su explotación a la ejecución secuencial y a programas que presentan el tipo de paralelismo considerado.

- **Reparto del tiempo** entre la actividades que desarrolla un procesador al explotar el tipo de paralelismo considerado.

Además, se presentan otros resultados particulares de cada tipo de paralelismo como la comparación con el método de explotación por copia en el caso del paralelismo_O, y el ahorro de tiempo en desreferenciaciones que supone la explotación del paralelismo_Y. Finalmente, se comparan los resultados obtenidos con los de otros sistemas.

Para realizar las medidas se han utilizado *benchmarks*, programas cuyo proposito es medir el rendimiento característicos de un sistema y que por ser comunmente utilizados permiten realizar comparaciones entre distintos sistemas. Un bechmark sintético es un programa simple que aproxima el rendimiento de un tipo de aplicación. Los bechnmarks utilizados, que han sido comentados brevemente en el capítulo 4, son programas muy conocidos. Las medidas se han realizado ejecutando tres veces cada programa.

6.2 Evaluación de la Explotación del Paralelismo O

La Figura 6.1 muestras las curvas de incremento del rendimiento para los programas considerados. La curva representa un incremento de velocidad lineal en programas con paralelismo de grano grueso como *queen10*, desciende al disminuir el grano del programa (*queen8*). Para el programa *chat*, que presenta un paralelismo de grano fino sólo se consigue una ligera mejora del rendimiento.

6.2.1 Reparto del tiempo

Para analizar los resultados obtenidos vamos ver como se emplea el tiempo de un procesador básico. Cuando se ejecuta un programa con paralelismo_O el tiempo de un procesador básico se reparte fundamentalmente entre las siguientes actividades:

- **Prolog**
  Tiempo empleado en la ejecución del programa.
- **Inactividad**
  Tiempo que el procesador ha estado desocupado. Este parámetro

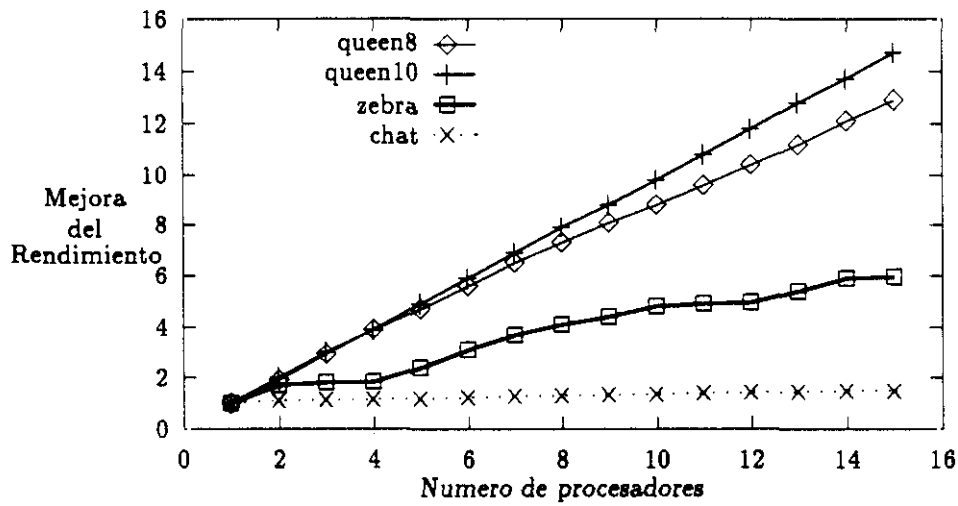


Figura 6.1: Mejora del rendimiento conseguida explotando Paralelismo_O

recoge el tiempo en que no hay trabajo en el sistema para el procesador, y también el tiempo que transcurre entre una petición de trabajo hasta la recepción del mismo, debido a las comunicaciones.

- **Recomputación**
  Tiempo empleado en recomputar el camino de éxito recibido de otro *procesador*.

- **Comunicaciones**
  Tiempo dedicado a las comunicaciones con el controlador y con otros procesadores. Este tiempo incluye el empleado en la composición de los mensajes de envio de trabajo a otros procesadores.

El porcentaje de tiempo dedicado a cada actividad ha sido obtenido como la media de los porcentajes de cada uno de los procesadores que han tomado parte en una ejecución. La Tabla 6.1 presenta los resultados obtenidos. En todos los programas, el porcentaje de ejecución de programas, (*Prolog*), desciende, mientras el porcentaje de *Inactividad* se incrementa al aumentar el número de procesadores, ya que cada programa tiene una cantidad de paralelismo limitada y al aumentar el número de procesadores desciende la cantidad de trabajo que se asigna a cada uno. Al considerar programas más largos los porcentajes de comunicaciones y de recomputación se reducen, ya que al aumenta el tiempo de *Inactividad* que ocupa una mayor parte del tiempo total. En todos los casos se observa que el porcentaje de tiempo dedicado a la recomputación es menor del 0.5%. Este porcentaje se va incrementando ligeramente con el número de procesadores, ya que cuanto mayor este número más intercambios de trabajo se producen (como confirma el aumento en el porcentaje de comunicaciones) y por tanto hay que recomputar un mayor número de veces. En el programa *chat*, el elevado porcentaje de *inactividad* indica el escaso paralelismo de este programa compardo con el anterior.

6.2.2 Comparación con el modelo por copia

Analizaremos los resultados obtenidos con los dos modelos de explotación del paralelismo_O considerados. La tabla 6.2 presenta estos resultados. Observamos que con un número pequeño de procesadores(4) el modelo por copia proporciona mejores resultados que el de recomputación, a excepción

Actividad	4 proc.	8 proc.	12 proc.	15 proc.
queen8				
Prolog	91.43	91.1	81.81	79.54
Inactividad	7.29	7.12	15.8	16.37
Recomputación	0.04	0.14	0.24	0.32
Comunicaciones	1.24	1.64	2.15	3.77
queen10				
Prolog	99.85	99.5	99.01	98.17
Inactividad	0.09	0.27	0.76	1.48
Recomputación	0.009	0.02	0.03	0.05
Comunicaciones	0.05	0.12	0.18	0.30
chat				
Prolog	7.61	6.45	4.32	3.12
Inactividad	91.9	93.1	95.2	96.3
Recomputación	0.01	0.01	0.01	0.02
Comunicaciones	0.41	0.43	0.46	0.52

Tabla 6.1: Reparto del tiempo de un procesador básico al explotar el paralelismo_O

del programa *queen10*, que tiene mayor paralelismo y trata con datos mayores. Al aumentar el número de procesadores, el modelo por recomputación proporciona mejores resultados, aunque en todos los programa la mejora del rendimiento conseguida por ambos métodos es muy similar.

programa	4 proc.		8 proc.		15 proc.	
	copia	recomp.	copia	recomp.	copia	recomp.
query	2.8	2.6	3.0	2.9	3.4	3.4
zebra	1.9	1.8	3.6	3.7	3.1	3.9
mm	2.2	2.1	3.4	3.4	4.5	4.5
queen(8)	3.1	2.8	7.4	7.3	12.9	12.9
queen(9)	2.4	2.4	7.7	7.7	14.1	14.2
queen(10)	3.0	3.1	8.0	7.9	14.0	14.7

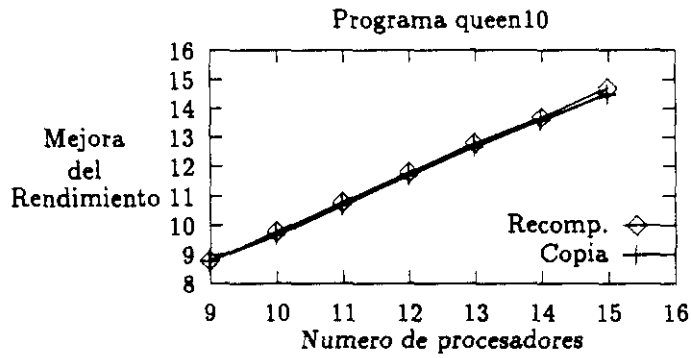
Tabla 6.2: Mejora del Rendimiento conseguida explotando paralelismo_O por copia y recomputación

La Figura 6.2 compara las curvas de incremento del rendimiento para los programas *queen8* y *chat*. Para el primero las curvas obtenidas por ambos métodos son muy similares y solo al aumentar por encima de 10 el número de procesadores se desajustan ligeramente a favor de la recomputación. Para el programa *chat* el desajuste es importante al aumentar el número de procesadores.

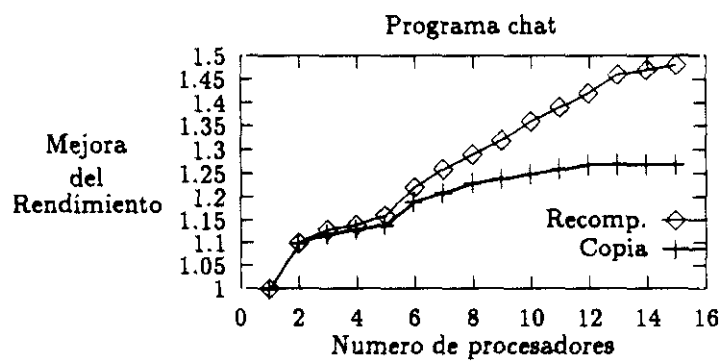
Para analizar estos resultados, se ha medido la forma en que reparten su tiempo los procesadores en el modelo por copia. Las actividades son las mismas que en el modelo por recomputación, excepto la recomputación que no se realiza en este caso. Observamos que a pesar de ahorrarse el tiempo de la recomputación, los tiempos dedicados a la ejecución de programas (*Prolog*) son ligeramente menores que el modelo por recomputación, ya que han aumentado los tiempos de comunicaciones.

6.2.3 Recargo introducido a la ejecucion secuencial

Para evaluar el recargo que la explotación del paralelismo introduce a la ejecución secuencial se ha medido el porcentaje de recargo introducido en la ejecución de programas sin explotar paralelismo en la máquina paralela sobre la ejecución en la máquina secuencial a partir de la que se ha desarrollado



a)



b)

Figura 6.2: Comparación del modelo por copia y recomputación

Actividad	4 proc.	8 proc.	12 proc.	15 proc.
queen8				
Prolog	94.18	90.91	81.0	76.49
Inactividad	3.44	6.28	13.52	18.11
Comunicaciones	2.38	2.81	5.55	5.4
queen10				
Prolog	99.77	99.5	98.9	98.09
Inactividad	0.13	0.22	0.73	1.19
Comunicaciones	0.12	0.19	0.37	0.72

Tabla 6.3: Reparto del tiempo de un procesador básico al explotar el paralelismo_O por copia

PDP. Los resultados se presentan el la Tabla 6.4.

Programa	Modelo por Copia	Modelo por Recomputación
queen(8)	4.2	4.33
queen(9)	6.7	6.8
queen(10)	12.2	12.5
zebra	0.4	0.4
chat	0.7	0.8

Tabla 6.4: *Recargo al tiempo de ejecución secuencial por la explotación del* Paralelismo_O

Se observa un pequeño porcentaje de recargo (alrededor del 1%) introducido por el modelo por copia debido principalmente a la comprobación de la llegada de mensajes. Por otra parte comparando el recargo del modelo por copia con el del modelo por recomputación obtenemos el recargo (menor del 0.1%) introducido por la anotación del camino de éxito (única diferencia entre ambos modelos cuando se ejecutan programas sin explotar paralelismo). *Este recargo puede evitarse anotando en el código del programa de entrada*

si el programa tiene paralelismo y por tanto es necesario registrar su camino de éxito.

6.2.4 Paralelismo Medio

Un parametro importante en un sistema multiprocesador es el *paralelismo medio* relacionado con la utilización de los procesadores. El *grado de paralelismo* (GP) de un programa se define como el número de procesadores utilizados en cada unidad de tiempo para su ejecución. La representación de GP como una función del tiempo proporciona el *perfil de paralelismo* de un programa. El *paralelismo medio* se define como

$$A = \frac{1}{t_2 - t_1} \int GP(t)dt$$

que en forma discreta puede expresarse como:

$$A = \frac{\sum_{i=1}^{m} i.t_i}{\sum_{i=1}^{m} t_i}$$

donde t_i es la cantidad total de tiempo que $GP = i$ y

$$\sum_{i=1}^{m} t_i = t_2 - t_1$$

es el periodo de tiempo total.

Para medir el paralelismo medio, ha sido necesario sincronizar los relojes de todos los procesadores del sistema. Cada procesador anota en una tabla el instante de tiempo en que cambia de estado. Cada vez que se envia una nueva respuesta al controlador se envia también la tabla de actividades que es almacenada en un fichero. A partir de estos datos se obtiene la gráfica correspondiente al perfil de paralelismo. La figura 6.3 muesta el perfil para el programa *queen10*. Se observa que el programa presenta un alto grado de paralelsimo y que se dá urante todo el tiempo de ejecución. El paralelismo medio para este programa es de 14.2, lo que indica que todos los procesadores estan ocupados durante toda la ejecución.

6.3 Evaluación de la Explotación del paralelismo Y

La gráfica de la Figura 6.4 muestra el rendimiento conseguido en PDP explotando el paralelismo_Y de los programas *merge*, *qsort* y *matriz*. Progra-

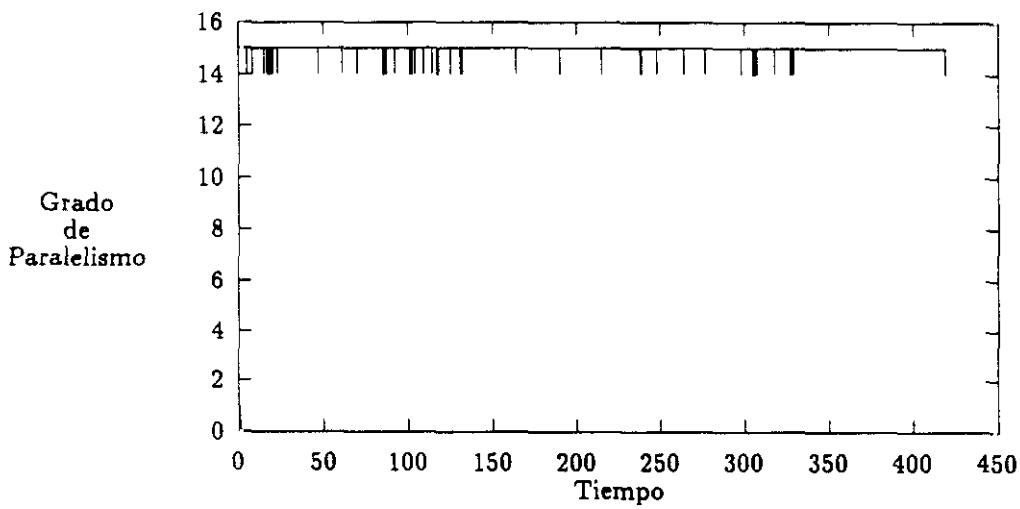


Figura 6.3: Perfil de paralelismo del programa queen10

mas cuyo paralelismo es de grano más fino como el que calcula los números de Fibonacci o el de las torres de Hanoi, no mejoran el rendimiento al explotar su paralelimo en PDP. Las curvas de mejora del rendimiento muestran que la cantidad de paralelismo de los programas es pequeña (el tiempo de ejecución se hace constante a partir de 10 procesadores) a pesar del tamaño de los datos, ya que PDP sólo mejora el rendimiento para programas con paralelismo de grano muy grueso.

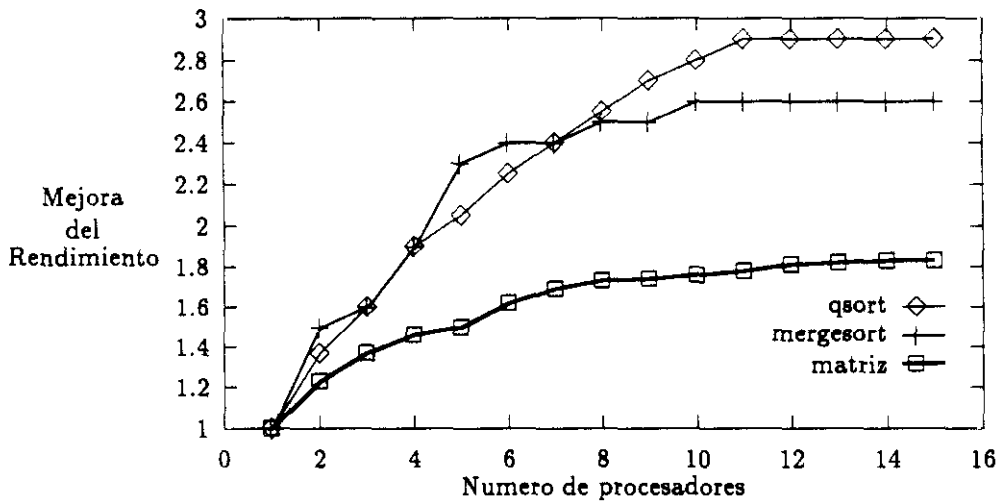


Figura 6.4: Mejora del rendimiento conseguida explotando Paralelismo_Y

6.3.1 Reparto del tiempo

Cuando se ejecuta un programa con paralelismo_Y el tiempo de un procesador básico se reparte fundamentalmente entre las siguientes actividades:

- **Prolog**
  Tiempo empleado en la ejecución del programa.
- **Inactividad**
  Tiempo que el procesador ha estado desocupado.

Actividad	4 proc.	8 proc.	12 proc.	15 proc.
qsort				
Prolog	72.04	54.84	42.24	30.09
Inactividad	14.02	32.23	46.85	61.02
Espera	7.93	7.12	6.85	4.39
Comunicaciones	6.01	5.81	4.06	4.5
merge				
Prolog	43.08	26.16	15.0	12.25
Inactividad	43.82	62.58	77.78	80.88
Espera	6.95	5.12	3.34	1.73
Comunicaciones	6.15	6.14	3.88	5.14
matriz				
Prolog	39.75	23.75	17.38	12.06
Inactividad	56.11	72.34	79.50	84.87
Espera	0.11	0.11	0.11	0.11
Comunicaciones	4.03	3.8	3.01	2.96

Tabla 6.5: Reparto del tiempo de un procesador básico al explotar el paralelismo_Y

- **Espera**
  Tiempo de inactividad debido a la espera de respuestas de otros procesadores.

- **Comunicaciones**
  Tiempo dedicado a las comunicaciones con el controlador y con otros procesadores. Este tiempo incluye el empleado en la composición de los mensajes de envio de trabajo a otros procesadores.

La Tabla 6.5 muestra los resultados obtenidos. Se observa que el porcentaje de tiempo de *inactividad* es muy elevado, y muy superior al que se obtenía en la explotación del paralelismo_O. Esto se debe a que la cantidad de paralelismo_Y de grano grueso presentada por estos programas es menor que el paralelismo_O de los programs considerados en su caso. También se observa que el tiempo dedicado a las *comunicaciones* es mayor que en el caso

de paralelismo_O por lo que la mejora del rendimiento obtenida es menor que aquel caso. El tiempo de *espera* es mayor cuanto mayor es el tiempo dedicado a la ejecución del programa (*Prolog*), ya que es cuando se ejecutá el programa, cuando se explota el paralelismo y se producen esperas de las respuestas.

6.3.2 Recargo introducido a la ejecucion secuencial

La Tabla 6.6 presenta el porcentaje de recargo (alrededor del 15%) introducido al la ejecución de programas sin explotar el paralelismo en sistema que explota paralelismo_Y. Este recargo se debe al tratamiento de las comunicaciones, a la comprobación sobre el modo del sistema (secuencial/paralelo), a las extensiones de procedimiento de backtracking para considerar las nuevas estructuras de tratamiento del paralelsimo y a extensiones similares en las instruccione de la máquina.

Programa	Porcentaje de Recargo
qsort(700)	17
merge(500)	15.62
matriz(75)	15.06

Tabla 6.6: Recargo al tiempo de ejecución secuencial por la explotación del Paralelismo_Y

6.3.3 Ahorro en el número de desreferenciaciones

En la WAM, la vinculación de una variable a un término se implementa como una referencia a dicho término. Las variables pueden pasar a través de diversos niveles de llamada a procedimientos, creandose una cadena de vinculaciones. Por tanto, la unificación de una variable puede requerir una desreferenciación para encontrar el término al que está unificado. Aunque una desreferenciación no es muy costosa, el número de ellas puede ser tan elevado que constituyen un parametro fundamental del rendimiento del sistema. PDP reduce considerablemete el número de referenciaciones respecto al sistema secuencial, ya que cuando se envian los objetivos de una llamada paralela, se envian los términos desreferenciados y así se almacenan en el

Programa	1 proc.	4 proc.	8 proc.	15 proc.
merge (500)	39353	37918	36806	36584
qsort (700)	67462	65285	62267	61120
matriz (75)	28500	21252	17930	17326

Tabla 6.7: Número de desreferenciaciones

destino, de forma que cada vez que se desreferencian se consigue un ahorro respecto a la representación que tenian en el porcesador padre. La tabla 6.7 muestra la reducción en el número de desreferenciaciones para los programas considerados. El ahorro consegido en el número de desreferenciaciones va desde un 7% en el programa *merge* hasta un 39% en el programa *matriz*.

6.3.4 Paralelismo Medio

Al igual que en el caso de la explotación del paralelismo_O, para conseguir esta medida, los procesadores anotan los cambios de estado en una tabla. Pero como en este caso solo un un procesador de los que intervienen en la obtención de una solución la comunica al controlodor ha sido necesario implantar un mecanismo para que los restantes envien su tabla de actividades. Cuando el controlador recibe una solución envia un mensaje de petición de información a todos los procesadores de su grupo, que contestan enviando la tabla de actividades. La Figura 6.5 muestra el perfil de paralelismo para el programa *qsort*. Observamos que el paralelismo se concentra en una zona de tiempo, quedando procesadores desocupados en otros momentos. El paralelismo medio de este programa es de 10.5 procesadores.

6.4 Evaluación de la explotación del Paralelismo Combinado

La Tabla 6.8 presenta la mejora del rendimiento con 15 procesadores de los dos *benchmarks* sintéticos descritos en el capítulo 4, que presenta paralelismo Y_bajo_O y O_bajo_Y respectivamente. Se observa en ambos casos que el rendimiento ha mejorado respecto al obtenido explotando cada clase de paralelismo. La mejora del rendimiento obtenida supera la suma de la mejora debida a cada tipo de paralelismo. Esto se debe a que se reduce el tiempo

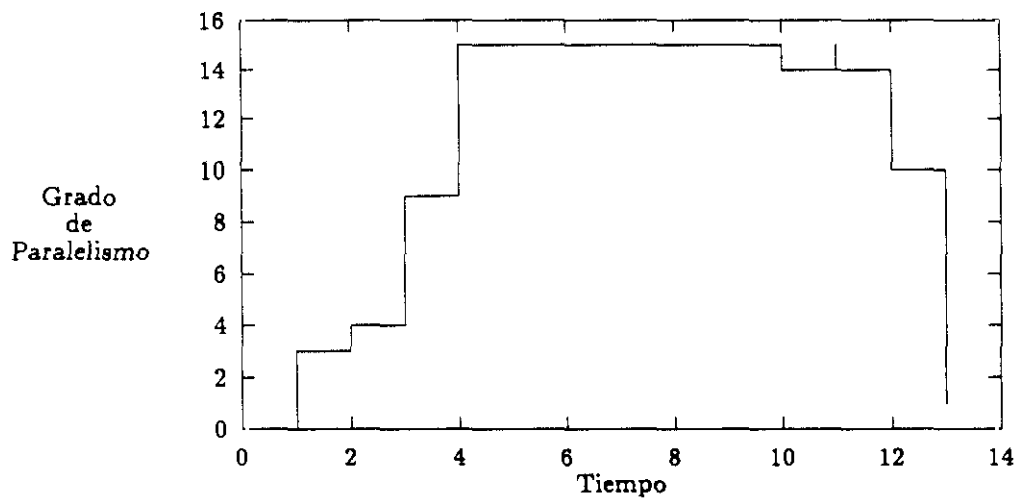


Figura 6.5: Perfil de paralelismo del programa qsort

program	OR_par.	AND_par.	Comb. par.
synthetic 1	1.5	2.9	4.5
synthetic 2	2.4	1.7	4.4

Tabla 6.8: Mejora del rendimiento por la explotación del Paralelismo Combinado

dedicado a las comunicaciones cuando se explota paralelismo O_bajo_Y, ya que el mecanismo de explotación del paralelismo_Y se transforma en el del paralelismo_O.

La figura 6.6 muestra las curvas de mejora del rendimiento para los benchmarks considerados.

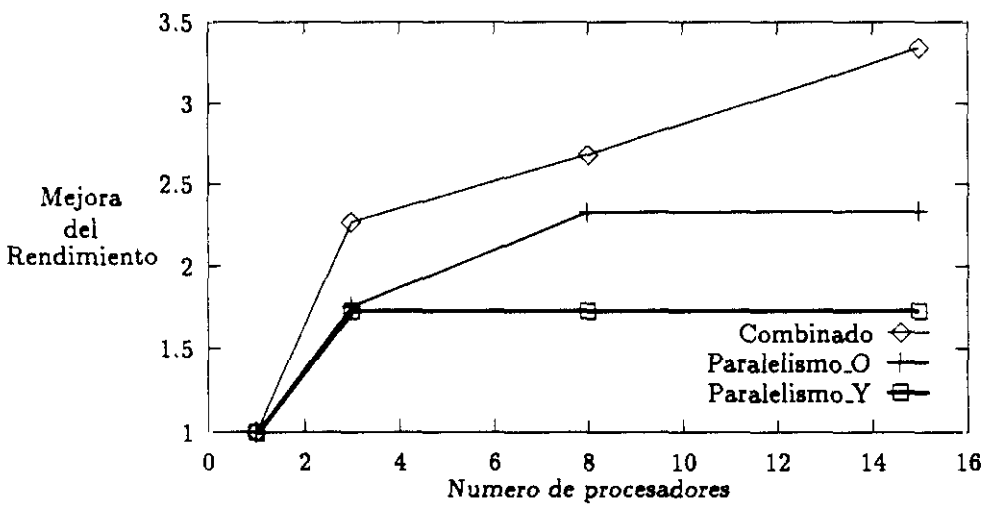


Figura 6.6: Speedup conseguida explotando ambos tipos de paralelismo

6.4.1 Recargo a la ejecución secuencial

La tabla 6.9 presenta el porcentaje de recargo introducido en programas secuenciales al ejecutarse en PDP. Este recargo, que se encuentra en torno

al 20% se debe a los mecanismos de explotación de cada tipo de paralelismo, y al mecanismo de combinación del paralelismo.

Programa	Porcentaje de Recargo
sintético1	19.3
sintético2	13.5

Tabla 6.9: Recargo al tiempo de ejecución secuencial

6.4.2 Recargo a la ejecución con Paralelismo_O puro

La tabla 6.10 presenta el recargo introducido a programas con paralelismo_O al ser ejecutados en PDP. Las medidas se han obtenido conparando el tiempo de ejecución en la máquina que explota únicamente el paralelismo_O, con el tiempo obtenido en PDP. Se observa que este recargo es pequeño, menor del 5% y se debe al mecanismo de explotación del paralelismo_Y.

Programa	Porcentaje de Recargo
sintético1	2.85
sintético2	0.88

Tabla 6.10: Recargo al tiempo de ejecución del sistema O_paralelo

6.4.3 Recargo a la ejecución con Paralelismo_Y puro

La tabla 6.11 presenta el recargo introducido a programas con paralelismo_Y al ser ejecutados en PDP. Las medidas se han obtenido conparando el tiempo de ejecución en la máquina que explota únicamente el paralelismo_Y, con el tiempo obtenido en PDP. El recargo medido es menor del 1%.

Programa	Porcentaje de Recargo
sintético1	0.49
sintético2	0.13

Tabla 6.11: Recargo al tiempo de ejecución del sistema Y_paralelo

6.5 Recargo de la planificación

En PDP el tiempo de los procesadores se distribuye entre las siguientes tareas:

- **Planificación:** Tiempo empleado por el controlador en realizar la planificación
- **Inactividad:** Tiempo que el procesador está desocupado
- **Comunicaciones:** Tiempo empleado en la recepción y envio de mensajes

Se observa que en todos los casos el tiempo de *inactividad* es muy elevado, o que indica que el número de procesadores que puden asignarse a un mismo controlador puede ser bastante mayor que 15. Los porcentajes son mayores en el programa *qsort* debido a que el tiempo total de ejecución es mucho menor que el de *queen10*.

6.6 Comparación con otros sistemas

Existen diversas implementaciones en el area de la explotación del paralelismo de Prolog. Sin embargo es dificil hacer una comparación directa con la nuestra debido a que se han evaluado con diferentes programas o tamaños de datos de entrada a estos programas así como que constan de distinto número de procesadores. La Figura 6.7a) muestra una comparación para el programa *qsort*. Verden [69] en un sistema distribuido para la explotación del paralelismo_Y, obtiene resultados similares a los de PDP para el programa *qsort*, aunque las pruebas se han realizado con una lista de

Actividad	4 proc.	8 proc.	12 proc.	15 proc.
qsort				
Planificación	6.76	7.2	6.8	7.1
Inactividad	91.14	89.2	84.6	80.22
Comunicaciones	2.1	3.6	8.6	12.68
queen10				
Planificación	1.2	1.9	7.6	11.2
Inactividad	98.6	97.1	90.6	86.4
Comunicaciones	0.20	1.04	1.72	2.36

Tabla 6.12: Reparto del tiempo del controlador

300 elementos. El esquema de Hermenegildo [33] sobre memoria compartida es lógicamente más eficiente al evitar las comunicaciones, sin embargo, al manejar programas que utilicen un número mayor de procesadores, los esquemas distribuidos pueden superar a los de memoria compartida. La Figura 6.7b) muestra una comparación para el programa *queen8*. La mejora del rendimiento obtenida para este programa es similar a la del sistema MUSE [2], pero al aumentar el número de procesadores utilizados por un programa, se superaía este rendimiento.

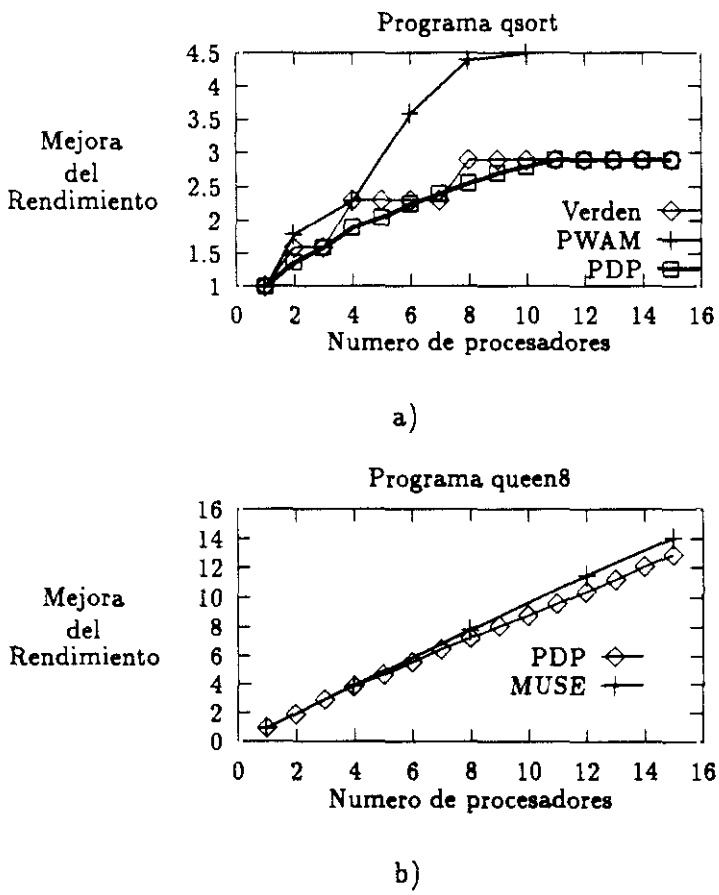


Figura 6.7: Comparación con otros sistemas

7

Conclusiones y Principales Aportaciones

7.1 Conclusiones

El sistema realizado permite extraer importantes conclusiones aplicables a la construcción de sistemas para la ejecución de Prolog. La explotación del paralelismo_O proporciona una mejora del rendimiento casi lineal, para programas con alta granularidad. Por el contrario, la explotación del paralelismo_Y puro obtiene resultados muy por debajo de los obtenidos en un sistema con memoria compartida. Esto se debe a dos causas principales. Por una parte el intercambio de trabajo por explotación del paralelismo_Y requiere un intercambio de mensajes mayor que en el caso del paralelismo_O. Por otra parte el grano de paralelismo_Y que presentan los programas suele ser menor que el de paralelismo_O. La aparición de paralelismo O_bajo_Y permite convertir el esquema de explotación del paralelismo_Y en el del paralelismo_O, obteniendo de esta forma las ventajas de éste y evitando el tráfico de mensajes que requiere la explotación del paralelismo_Y. Por tanto, los resultados apuntan a la conveniencia de recurrir a un sistema mixto en el que los trabajos surgidos del paralelismo_Y puro se repartan entre procesadores que comparten la memoria, mientras los procedentes del paralelismo_O u O_bajo_Y se reparten entre procesadores con memoria distribuida.

Se ha probado que el método de recomputación utilizado es viable en este tipo de sistema, aportando otras ventajas como la facilidad para realizar la combinación del paralelismo.

Una observación constante en la evaluación del sistema ha sido la necesi-

dad de controlar la granularidad de las tareas ejecutadas en paralelo. Sin estas medidas la explotación del paralelismo no sólo no mejora el rendimiento si no que puede empeorarlo. Se ha puesto de manifiesto la utilidad de técnicas heurísticas para realizar este control. El estudio de la planificación ha confirmado la importancia de la proximidad topológica.

El estudio hecho sobre la configuración de la red y la implementación sobre *transputers* ha permitido conocer las ventajas y problemas que presenta su utilización. La principal ventaja es la simplicidad de su esquema que permite estudiar con facilidad el comportamiento en las distintas situaciones que se presentan, sin el apantallamiento que se dá en sistemas más complejos, por ejemplo introduciendo un sistema operativo. Los principales inconvenientes ha sido las dificultades que presenta para la depuración de los y programas y para el sincronismo de la ejecución con la consecuente dificultad para la evaluación de los tiempos.

7.2 Principales Aportaciones

Las aportaciones de PDP son:

- Se ha realizado un estudio de los métodos de explotación del Paralelismo_O por reconstrucción del entorno de trabajo de la tarea padre: copia y recomputación. Los resultados han mostrado que la mejora del rendimiento alcanzada con el método de recomputación supera a la alcanzada con el de copia cuando el tamaño del sistema aumenta.

- Se ha realizado la explotación del Paralelismo_Y mediante entornos cerrados que permiten la computación autonoma de los objetivos independientes. La planificación paralela de estos objetivos se condiciona a su granularidad que se estima en función del tamaño de los argumentos. El cálculo de este tamaño es costoso pero el modelo de PDP lo porporciona automáticamente al construir el entorno cerrado correspondiente al objetivo, por lo que este control de granularidad no introduce recargo adicional en PDP. También se ha diseñado un mecanismo de backtracking inteligente que se aplica a los objetivos ejecutados por otras tareas.

- Se ha diseñado un método de combinación del paralelismo en el que la explotación del paralelismo O_bajo_Y se realiza coma la extensión

natural del modelo de recomputación del paralelismo_O. Además de la sencillez del modelo y la facilidad para su integración con los modelos de ejecución de cada tipo de paralelismo puro, mejora el tiempo de explotación del paralelismo_Y ya que cuando se presenta en forma de O_bajo_Y se producen computaciones autónomas.

- Se ha diseñado un esquema de control jerárquico del sistema en el que los controladores planifican el trabajo pendiente mientras que los restantes procesadores (básicos) seleccionan los trabajos que pueden ser computados en paralelo en base a estimaciones de granularidad. Se ha realizado un estudio de la planificación siguiendo distintoss criterios: balance de carga, antiguedad de los trabajos y proximadad topológica, resultando que este último proporciona la mejora del rendimiento mayor. Se ha realizado un estudio de la mejora del rendimiento obtenida introduciendo controles de granularidad basados en observaciones heurísticas.

- Se ha realizado una implementación de PDP en ANSI C paralelo sobre el sistema de transputers de un Supernodo Parsys con 18 transputers T800, tratando los problemas asociado: encaminamiento de mensajes, estudio de las situaciones de bloqueo, sincronismo y toma de medidas.

- Se ha realizado la evaluación de diversos aspectos de la explotación de cada tipo de paralelismo y su combinación: mejora del rendimiento, reparto del tiempo entre las distintas actividades que origina el paralelismo y el recargo que introduce a la ejecución de programas secuenciales.

7.3 Futuros Trabajos

Existen numerosos aspectos con los que continuar la exploración de técnicas que permitan mejorar el rendimiento de la ejecución algunos de los cuales se refieren al modelo de explotación y otros al soporte hardware:

- Implementación de los predicados de efectos laterales.

- Mejora de los tiempos absolutos de ejecución realizando una implementación que parta de un sistema secuencial que proporcione buenos tiempos absolutos, como SICSTUS.

- Realización de un sistema mixto que permita la aplicación de técnicas de memoria compartida en la explotación del paralelismo_Y y distribuida en los otros casos.
- Probar nuevas configuraciones de red, con un número mayor de elementos e investigar técnicas de simulación que permitan evaluar el sistema con alta precisión antes de realizar la implementación completa.
- Utilización de otros sistemas soportes como el nuevo transputer T9000.

A

Simulador de la Máquina Y Paralela

A.1 Introducción

Las investigaciones que tienen como objetivo mejorar el rendimiento de un sistema, requieren hacer predicciones de los niveles de rendimiento en las primeras etapas del desarrollo. Es común el uso de técnicas de simulación para la evaluación del rendimiento de la ejecución de los lenguajes lógicos [40] [17] [43]. Nosotros también comenzamos la implementación de la máquina Y paralela realizando una simulación sobre una estación de trabajo SUN en lenguaje C.

El simulador nos permitió la puesta a punto del modelo antes de enfrentarnos con la implementación real sobre transputers y las dificultades para la depuración que esta presentaba, proporcionando también las primeras estimaciones de los resultados.

Sin embargo la utilidad del simulador no terminó al construir la implementación real, ya que además del apoyo a la depuración que ha seguido suponiendo, nos permite hacer predicciones del comportamiento de extensiones del sistema real (mayor número de procesadores) o modificaciones del mismo (distintos compontes). Por otra parte estas predicciones son más fiables al haber validado los supuestos sobre los puntos críticos de la simulación y al haber ajustado los parametros del simulador en base a las medidas obtenidas en el sistema real. De esta manera, hay una realimentación entre los datos proporcionados por el silulador y el sistema real, que nos permite avanzar con seguridad en la implementación.

Los puntos que se han considerado en la simulación son los siguientes:

- *Implemetación del conjunto de máquinas Y paralelas*
  La implementación de las zonas de datos y del control de ejecución de cada procesador básico coincide con el real. Las diferencias se refieren únicamente al envio de mensajes
- *Implemetación del controlador*
  Coincide también con la implementación real salvo en el envio de mensajes
- *Simulación de la planificación de procesos*
  Es necesario simular la asignación de tiempo de CPU a cada máquina, de forma que el avance relativo de cada uno de ellos respecto al resto sigue el comportamiento real
- *Simulación de las comunicaciones*
  Hay que simular el envio de mensajes entre procesadores, teniendo en cuenta en tiempo empleado en la transmisión y su repercusión en el avance de la ejecución de los procesadores origen y destino

A.2 Simulación de la planificación de procesos

El paralelismo se simula mediante asignación rotatoria (round-robin) del tiempo de CPU a los diferentes procesadores, incluyendo el controlador (Figura A.1). En cada ciclo de simulación se ejecutan las instrucciones actuales de los procesadores activos y se tratan los mensajes que han cumplido un tiempo de espera igual al de latencia de la red de interconexión. Una simulación precisa del tiempo exige, sin embargo, un avance temporal idéntico para cada procesador, y lo suficientemente pequeño para garantizar la ausencia de adelantos que distorsionen la disponibilidad de recursos que el sistema tendría en una situación real. Evidentemente esto no se cumple si en cada ciclo de simulación se ejecuta una instrucción completa de la máquina abstracta, ya que los tiempos de éstas son diferentes y en general dependientes de los datos. Esto podría evitarse codificando las instrucciones de la máquina abstracta con instrucciones elementales tipo RISC (una por ciclo máquina) y utilizando como ciclo de simulación el ciclo máquina del RISC. Sin embargo, esta alternativa complicaría en exceso el desarrollo y limitaría su portabilidad a una arquitectura paralela real. La solución que hemos

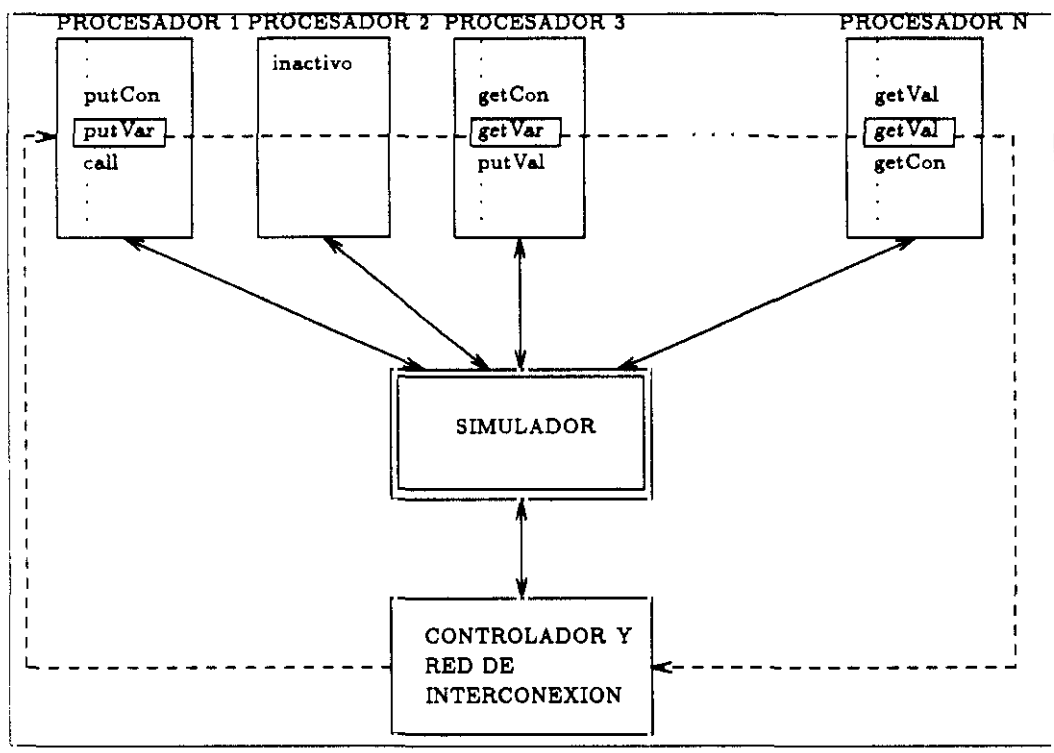


Figura A.1: Esquema del simulador

adoptado consiste en permitir que un procesador avance en cada ciclo de simulación una instrucción completa de la máquina abstracta, pero midiendo el tiempo de ejecución real empleado. La siguiente instrucción de este procesador no se ejecuta hasta llegar al ciclo de simulación en el que los demás procesadores activos hayan consumido un tiempo igual o superior, siguiendo el esquema de la Figura A.2.

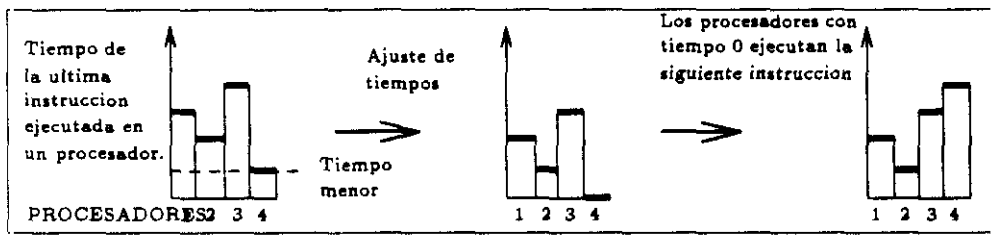


Figura A.2: Simulación del avance temporal

Esta solución es posible en un modelo de memoria distribuida porque las interacciones entre procesadores solo tienen lugar mediante paso de mensajes, y estos los temporiza el simulador con los valores estimados de la red física de interconexión. Cuando llega el turno de un procesador las zonas de memoria y registros de la máquina recuperan los contenidos que tenian cuando se ejecutó la última instrucción de dicho procesador. A continuación se leen los mensajes que ha recibido el procesador desde su última ejecución, se ejecuta la siguiente instrucción y se cede el turno al siguiente procesador activo. El proceso continua hasta que alguno de los procesadores llega al final del programa.

Una dificultad encontrada al realizar las medidas es la falta de precisión del reloj de la estación de trabajo, de manera que de una ejecución a otra de un mismo programa el resultado variaba significativamente. Para tener precisión en las medidas se ha prescindido del reloj de la estación, asignando un tiempo fijo a la ejecución de cada instrucción. Tambien se tiene en cuenta la dependencia del tiempo con los datos de entrada al considerar el tiempo empleado en cada paso por funciones recursivas como la desreferenciación o la unificación. En una primera fase estos tiempos fueron una estimación, siendo las medidas de tiempo en la máquina Y paralela relativas a las medidas en la máquina secuencial, también implementada en la estación. Al realizar la implementación sobre transputers estos tiempos se ajustaron a

las medidas reales.

A.3 Simulación de las comunicaciones

El envio de mensajes se simula copiando el mensaje del almacen de salida del procesador origen en el de entrada del procesador destino. Para simular el tiempo empleado en la transmisión se retarda dicha copia hasta que la ejecución en los procesadores activos avanza el tiempo equivalente al asignado a la transmisión.

Después de cada ciclo de ejecuciones de los procesadores, se actualiza el tiempo de las trasmisiones pendientes, decrementandolo en el tiempo consumido por los procesadores activos en el último ciclo de simulación. A continuación se copian los mensajes cuyo tiempo de transmisión ha llegado a 0.

De esta forma tenemos en cuenta el tiempo empleado en la transmisión pero falta considerar el retardo que supone el envio de mensajes en el procesador origen debido a la comunicación sincrona de los transputers. Para ello el tiempo de la ejecución de la instrucción que origina el envio se incrementa en el tiempo estimado del retardo.

A.4 Ajuste de tiempo

Según se ha descrito en las secciones anteriores la simulación se basa en el avance relativo de la ejecución en cada procesador. Los resultados de la simulación solo tendrán un valor absoluto si estan ajustados a las caracctérísticas de una implementación real.

Los tiempos que necesitamos conocer son:

- Tiempo de ejecución de cada instrucción
- Tiempo de tranmisión de mensajes
- Retardo que supone el envio de un mensaje para el procesador origen

En todos los casos se encuentra que la medida del tiempo de una solo dato (ya sea una instrucción o una transmisión) es del orden de la precisión del reloj del transputer, por lo que para reducir el error se ha considerado la media de los datos de las ejecuciones. Así se ha obtenido que el tiempo de

medio de ejecución de una instrucción es de 35 μs. Mientras que el tiempo medio de envio de un mensaje es de 60 μs y el retardo provocado en un procesador por el envio de un mensajes de de 30 μs.

A.5 Estructura del Simulador

El simulador consta de un único proceso que simula cíclicamente a cada procesador activo, ejecutando las instrucciones que correspondan a aquellos que no tengan pendiente tiempo de ejecuciones anterios. El ciclo se completa con la ejecución del las tareas del controlador. Después de cada ciclo de simulación se actualizan los tiempos pendientes de cada procesador, restando el menor de los tiempo de ejecución de las instrucciones del ciclo. Así mismo se actualizan loas tiempos pendientes de los mensajes enviados. Finalmente se simulan las comunicaciones, transmitiendo los mensajes que han completado su tiempo de ejecución. La estructura del simulador aparece en la Figura A.3.

Para la simulación se dispone de una variable *procesador_actual*, que en cada momento indica el procesador que se está activado. En la inicialización el contador de programa (IC) del procesador que se elija como inicial se carga con la dirección de comienzo de ejecución, el estado del procesador se pone ACTIVO y los punteros y zonas de memoria se inicializan a los que corresponden a este procesador. Una vez que se ha iniciado la ejecución cada procesador cuando llega su turno lee los mensajes que pudira tener pendientes y que podrian cambiar su estado por ejemplo de ESPERA a ACTIVO y a continuación si está ACTIVO ejecuta la siguiente instrucción, apuntada por su contador de programa. Si no está ACTIVO envia una peticin de trabajo al controlador, pasando a estado de ESPERA. Cada vez que se ejecuta una instrucción se conmuta al siguiente procesador haciendo que las zonas de memoria sean las correspondientes al procesador al que ha llegado el turno.

```
/* bucle principal */
while not fin_programa do
begin
   trata_mensajes(procesador_actual);
   /* Si el procesador está inactivo pide trabajo */
   if estado[procesador_actual] = INACTIVO then
   begin
      enviar_peticion(procesador_actual);
      estado[procesador_actual] := espera;
   end
   /* Si el procesador está activo y sin tiempo pendiente */
   /* se ejecuta la siguiente instrucción */
   if estado[procesador_actual] = ACTIVO and tiempo[procesador_actual] = 0 then
      tiempo[procesador_actual] = ejecutar_instruccion();
   /* Se selecciona el siguiente procesador */
   salvar_estado(procesador_actual);
   procesador_actual := (procesador_actual + 1) mod NUM_PROCESADORES
   /* Se comprueba si se ha completado una ronda */
   if procesador_actual = 0 then
   begin
   /* Se actualizan los tiempos pendinetes restando el menor */
      actualizar_tiempos();
      /* Se simula el controlador */
      controlador();
      /* Se actualizan los tiempos de transmisión de mensajes */
      actualizar_mensajes();
      /* se simulan las comunicaciones */
      comunicar();
   end
   /* se conmuta al siguiente procesador */
   cambiar_estado(procesador_actual);
end
```

Figura A.3: Esquema del simulador

B

Predicados Extralogicos

B.1 Introducción

Uno de los objetivos de PDP es ejecutar programas Prolog con su semántica estandard, suportando también los predicados con efectos laterales como el *corte*, *findall*, los predicados de entrada/salida y los de manejo de la base de datos interna. Aunque aún no han sido implementados todos ellos, se ha realizado el diseño que se presenta en este apendice.

La forma más sencilla de mantener la semántica secuencial de estos predicados con efectos laterales es permitir su ejecución paralela solo cuando la rama en que se encuentran es la más a la izquierda del árbol de búsqueda. Para incluir estos predicados en el sistema es necesario incluir mecanismos eficientes que permitan comprobar si la rama actual está a la izquierda del árbol.

La ejecución de los predicados con efectos laterales se simplifica en PDP suponiendo una transformación del programa en tiempo de compilación que simplifica su analisis en tiempo de ejecución.

B.2 Corte

La mayor parte de los programas Prolog prácticos contienen cortes. El corte suele ser utilizado para mejorar la eficiencia de un programa al evitar la consideración de determinadas soluciones alternativas. La semantica del corte que presentan la mayoria de los sistemas consiste en que *la operación de corte fija todas las elecciones hechas desde que se invicó el predicado que*

contenia el corte, eleiminando otras posibles elecciones.

B.2.1 Implementación secuencial

El mecanismo de PDP para el tratamiento del corte supone la transformación del programa propuesta por Van Roi [68]. Los predicados que contienen un corte comienzan con una llamada al predicado interno *cut_load(X)*. Este predicado carga X con el valor del registro *BReg* que apunta al último punto de elección. El resto del cuerpo del predicado consiste en una llamada al predicado transformado al que se para el valor de X. El predicado transformado se diferencia del original en que se ha añade X como parametro a la cabeza de cada cláusula y en que cada corte se sustituye por una llamada al predicado interno *cut*. Este predicado carga *BReg* con el valor de X, restaurando el valor del último punto de elcción de la pila de control. Por ejemplo, el siguiente fragmento de programa:

```
(1)   p :- q.
(2)   p :- r.
(3)   p :- s, !, t, !, u.
(4)   p :- v.
```

se transforma en:

```
p :- cut_load(X), p'(X).

p'(X) :- q.
p'(X) :- r.
p'(X) :- s, cut(X), t, cut(X), u.
p'(X) :- v.
```

B.2.2 Implementación Paralela

En implementaciones secuenciales (WAM) un solo procesador se encarga de explorar todas las cláusulas alternativas correspondientes a las ramas del árbol de búsqueda que es explorado en profundidad y de izquierda a derecha. Un corte en la rama izquierda de un punto de elección se encuentra siempre antes que uno de las ramas de la derecha. Por lo tanto es sencillo y eficiente encontrar y eliminar las ramas derechas. Cuando se explota el

paralelismo_O de Prolog las ramas alternativas con corte de un mismo predicado pueden ser procesadas por distintos procesadores y por tanto puede ocurrir que el proceso que se encarga de una rama con corte que no es la más izquierda encuentre dicho corte antes que el procesador encargado de la rama con corte más izquierda. El sistema debería eliminar todas las ramas a la derecha de la primera con corte empezando por la izquierda. Una implementación paralela realizada de esta forma requeriría anotar que procesos son los encargados de cada rama para poder pararlos si fuese necesario. El mecanismo resultaría caro ya que se realizaría trabajo innecesario. Otra posibilidad [32] consiste en suspender la ejecución de una rama con corte hasta que se encuentra a la izquierda del árbol. Otra posibilidad [6] que es la adoptada en PDP, es restringir la explotación del paralelismo, a los casos en que el corte puede manejarse eficientemente. En el ejemplo del apartado anterior se explorarían en paralelo las cláusulas (1), (2) y (3). La (4) solo se exploraría en paralelo con las anteriores si la 3 fallase antes de encontrar el corte. Para ello introducimos los nuevos predicados internos *begin_cut* y *end_cut* que marcan las cláulas que contienen algún corte, haciendo que la máquina entre en un nuevo modo de funcionamiento en el que no se explota el paralelismo. El código del ejemplo anterior pasa a ser:

```
     p :- cut_load(X), p'(X).

(1)  p'(X) :- q.
(2)  p'(X) :- r.
(3)  p'(X) :- begin_cut, s, cut(X), t, cut(X), end_cut, u.
(4)  p'(X) :- v.
```

Cuando se ejecuta la instrucción *cut_load* se carga en X la posición del último punto de elección. Se explota el paralelismo_O hasta encontrar una instrucción que comience con *begin_cut*. A partir de entonces deja de explotarse el paralelismo hasta que se produce un fallo o aparece la instrucción *end_cut*. En el ejemplo se explotaría el paralelismo del predicado u o se si produjese en la ejecución del predicado s, se explorarian en paralelo la cláusula (4) y otras posibles cláusulas de p'.

B.3 Negación como fallo Finito

Se implementa mediante un conmutador de la máquina que el procedimiento de fallo consulta antes de informar del fallo. Para ello se introducen los predicados internos *begin_not* y *end_not* mediante los que el compilador señala el objetivo negado. Por ejemplo el siguiente fragmento de programa:

```
p :- not(q).
```

se transforma en:

```
p :- begin\_not, q, end\_not.
```

B.4 Fallo

Se implementa mediante el predicado interno *fail* que provoca una llamada al procedimiento de fallo.

B.5 Findall

Este predicado, cuya sintaxis es *findall(Varible, Objetivo, Coleccion)*, recoge en la lista *Coleccion* todas las instancias de *Variable* correspondientes a todas las pruebas del *Objetivo*. En los sistemas Prolog secuenciales el orden de las soluciones recogidas está determinado por la estrategía de recorrido del árbol en profundidad y de izquierda a derecha. El los sistemas Prolog que explotan Paralelismo_O las soluciones se alcanzan en un orden arbitrario. Algunos sistemas como Muse [2] [5] preservan el oreden de las soluciones. Otros [15] como en el caso de PDP no lo hacen por razones de eficiencia.

PDP implementa *bagof* como un predicado interno que causa que al explotar el paralelismo_O del *Objetivo* de *findall* se envie a los procesadores que comparten este trabajo un mensaje especial que les indica que al alcanzar la solución no deben enviarla a la entrada/salida, si no al procesador padre. Si estos procesadores a su vez vuelven a explotar paralelismo_O indican a los nuevos procesadores que comparten el trabajo quien es el procesador padre. Queda por resolver el problema de que el procesador padre necesita saber cuando se han recibido todas las soluciones. Para ello los procesadores que han compartido el trabajo informan al padre del fallo al que llegan cuando han explorado todas las alternativas de que se encargaban.

B.6 Operadores Aritméticos y Logicos

Las operaciones aritméticas y lógicas se trasnforma en predicados internos de la siguiente forma:

```
a(X,Y,Z) :- Z is X + Y.
a(X,Y,Z) :- Z is X - Y.
a(X,Y,Z) :- Z is X * Y.
a(X,Y,Z) :- Z is X / Y.
a(X,Y) :- X = Y.
a(X,Y) :- X <> Y.
```

Se *transforman en:*

```
a(X,Y,Z) :- suma(X, Y, Z).
a(X,Y,Z) :- resta(X, Y, Z).
a(X,Y,Z) :- mult(X, Y, Z).
a(X,Y,Z) :- divi(X, Y, Z).
a(X,Y) :- igual(X, Y).
a(X,Y) :- distinto(X, Y).
```

B.7 Predicados de Entrada/Salida

Puesto que no mantenemos el orden de las soluciones en la ejecución secuencial, los predicados de entrada/salida se tratan cuando aparecen en cada rama. Para facilitar al usuario la comprensión de la ejecución, cada solución obtenina se acompaña de la lista de datos leidos que le corresponde. En PDP la entrada/salida solo puede realizarse desde el procesador que está directamente conectado al *host*. Por ello cuando apacece una de estos predicados por primera vez en una rama, es enviado al host para ser ejecutado allí. Cuando al explotar_paralelismo_O un procesador está *recomputando* una rama con un predicado de entrada/salida, no repite la operación, y si se trata de una lectura recibe del procesador padre el dato leido en la primera computación de la rama.

B.8 Predicados que modifican la base de datos

Solo tienen efecto en la siguiente ejecución, después de una nueva compilación.

C

Analisis del Protocolo de Comunicaciones

C.1 Introducción

Un cuidadoso diseño y analisis del protocolo de comunicaciones es imprescindible cuando el sistema alcanza cierta complejidad. La utilización de métodos formales para esta tarea permite eliminar la imprecisión de las descripciones informales, realizar demostraciones formales de la validez de un protocolo y utilizar computadores en el proceso de diseño y validación. De los numerosos métodos propuestos para la validación de protocolos [54] [53] [37] nosotros hemos adoptado el propuesto por Zafiropulo *et al.* [77] para realizar una validación automática del protocolo que ya estaba en un avanzado estado de desarrollo. El método usado está basado en el analisis de alcanzabilidad y utiliza una técnica de perturbaciones.

C.2 Tipos de errores de Diseño

Supondremos que los procesadores está inicializados correctamente antes de comenzar las interacciones. En este marco, podemos tratar cuatro tipos de errores potenciales de diseño: *estados de bloqueo*, *recepciones no especificadas*, *interacciones no ejecutables* y *estados ambiguos*.

- **Estado de bloqueo**
  Aparece cuando todos y cada uno de los procesos tiene como única

alternativa permanecer indefinidamente en el mismo estado. Los estados de bloqueo generalmente representan errores, pero pude haber excepciones. Pueden diseñarse protocolos que terminan en estado sin salida al completar su función.

- **Recepción No Especificada**
  Se produce cuando un arco etiquetado con un entero positivo no ha sido especificado en el diseño. Las recepciones no especificadas son peligrosas porque pueden llevar al proceso receptor a un estado desconocido en que tendrá un comportamiento imprevisible. Los protocolos pueden protegerse mediante un mecanismo que considere toda recepción no especificada. Sin embargo, si se introduce este mecanismo, las recepciones no especificadas que no sean causadas por un mal funcionamiento será tratadas de la misma forma. Por ejemplo, si la petición de una nueva conexión se hace mediante un nuevo identificativo, se tratará de una recepción no especificada pero correcta.

- **Interaciones No Ejecutables**
  Se producen cuando el diseño incluye transmisión y recepción de mensajes que no pueden producirse bajo condiciones normales de operación. Una interacción no ejecutable es equivalente a la existencia de código muerto en un programa. Pueden deberse a errores de diseño o a la consideración de condiciones de funcionamiento anormales. Para distinguir entre la condiciones normales y anormales, es una buena práctica de diseño, diseñar y validar un protocolo para la operatividad normal antes de introducir el tratamiento de la excepciones.

- **Estados Ambiguos**
  Se produce cuando un estado de un proceso puede cohexistir de forma estable con diferentes estado de otro proceso. No representan necesariamente un error.

C.3 Analisis de Perturbaciones

El analisis de perturbaciones es una técnica para detectar la presencia de potenciales errores de diseño en un protocolo. Cada estado del sistema se representa por un array de dimensión igual al número de procesos que interaccionan cuyos elementos de la diagonal principal representan el estado de

un proceso P_i y cada elemento i, k no perteneciente a la diagonal representa el mensaje enviado por el proceso p_i al proceso p_k.

$$\begin{array}{c} ESTADODEL \\ SISTEMA \\ (SS) \end{array} \left(\begin{array}{ccc} \begin{array}{c} ESTADO \\ CONTROL \end{array} & \begin{array}{c} C \rightarrow W0 \\ CANAL \end{array} & \begin{array}{c} C \rightarrow W1 \\ CANAL \end{array} \\ \\ \begin{array}{c} W0 \rightarrow C \\ CANAL \end{array} & \begin{array}{c} ESTADO \\ W0 \end{array} & \begin{array}{c} W0 \rightarrow W1 \\ CANAL \end{array} \\ \\ \begin{array}{c} W1 \rightarrow C \\ CANAL \end{array} & \begin{array}{c} W1 \rightarrow W0 \\ CANAL \end{array} & \begin{array}{c} ESTADO \\ W1 \end{array} \end{array} \right)$$

C: control W0(1): procesador básico 0(1)

El proceso comienza definiendo el estado inicial SS0 que consiste en los procesos en estado $S0$ y con los canales vacios (representado por E). SS0 se *perturba* a todos los posibles estado sucesores *alcanzables* ejecutando una única transmisión en uno de los procesos individuales $P_1, P_2, .., P_n$. Así, cada una de la posibles transmisiones a partir de SS0 genera una rama de un árbol llamado de *alcanzabilidad*. El procedimiento continua perturbando cada uno de los nuevos estados del sistema y termina cuando no se crean nuevos estados del sistema. Una vez que se ha generado el árbol de alcanzabilidad, los errores de diseño se detectan de la siguiente forma:

- **Estado de bloqueo**
  Los bloqueos se identifican en el árbol de alcanzabilidad por estados que tienen todos los canales vacios y no tiene transiciones de salida.

- **Recepción No Especificada**
  Se identifican por estados sin transiciones de salida que absorban el mensaje pendiente de transmisión de un canal.

- **Interaciones No Ejecutables**
  Se identifican por transiciones de estado presentes en el diseño que no aparecen en el árbol de alcanzabilidad.

- **Estados Ambiguos**
  Se identifican por el estado de un proceso concreto que aparece en diversos estados del sistema.

mensaje	numero
CODIGO	1
PETICION	2
TRABAJO	3
CONFIRMACION	4
INTERCAMBIO	5
NEGACION	6
RESPUESTA	7
BORRAR	8
NUEVO_TRABAJO	9

Tabla C.1: Numeración de los mensajes de PDP

C.4 Modelización del protocolo de PDP

Para realizar la validación del protocolo vamos a modelizarlo utilizando una simplificación de la representación dada en el capítulo 5, en la que los procesos que interaccionan se modelan por un grafo de estados finito. Los mensajes intercambiados entre los procesos se representan por enteros. La emisión de mensajes se representa por valores negativos del entero correspondiente y la recepción por su valor positivo. La Tabla C.1 y C.2 representan la numeración de los mensajes de PDP y de sus estados, respectivamente. Para realizar el analisis se han considerado tras procesos: el controlador y dos procesadores básicos. De esta forma aparece toda la gama de mensajes y situaciones posibles en el sistema. El analisis se ha realizado sobre el protocolo de comunicaciones correspondiente a la explotación del paralalismo_Y, ya que el del paralelismo_O es una simplicficación de este.

La forma en que queda representado el protocolo de comunicaciones de PDP aparece en la Figura C.1. Se han introducido algunos estados ficticios, como el *tratamiento de peticiones* (Tarta_pet) en el diagrama que representa los intercambios entre el controlador y un procesador básico, para ajustarse al esquema del método y evitra recibir y enviar mensajes desde un mismo estado. De la misma forma en el protocolo entre procesadores básicos se han introducido los nuevos estados de *recepción de trabajo* (Recv_tra), *activo con petición* (Act_pet), *activo con trabajo* (Act_tra), *recepción de la orden de intercambio* (Recv_int) y *recepción de respuesta* (Recv_res).

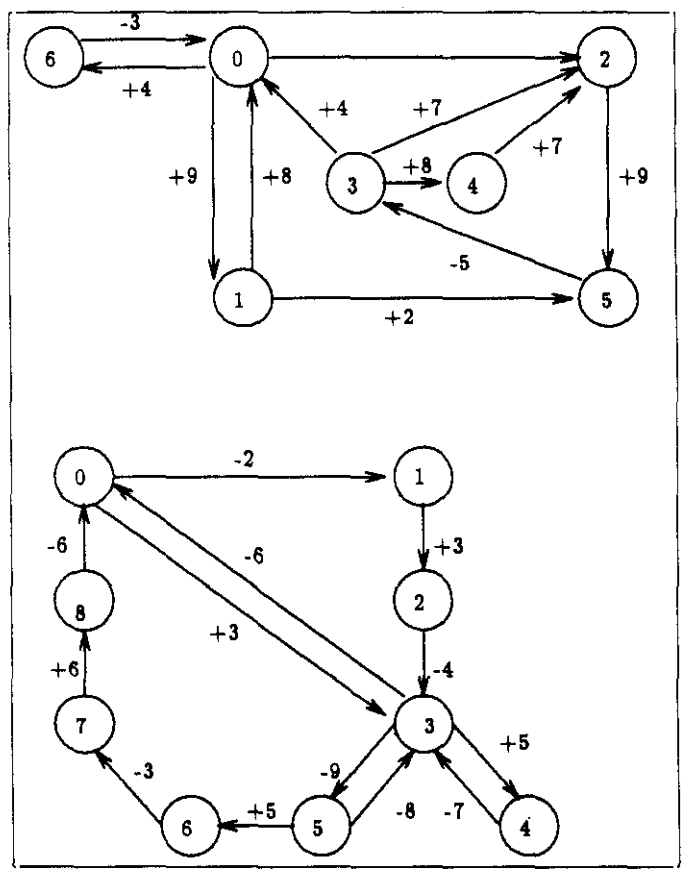


Figura C.1: Protocolo de comunicaciones de PDP

Controlador		Pro. básico	
estado	numero	estado	numero
Libre	0	Libre	0
Esp_pet	1	Esp_tra	1
Esp_tra	2	Recv_tra	2
Esp_conf	3	Activo	3
Esp_neg	4	Act_pet	4
Trata_pet	5	Act_tra	5
Tra_ini	6	Recv_int	6
		Esp_resp	7
		Recv_resp	8

Tabla C.2: Numeración de los estados

C.5 Resultados obtenidos

Se ha implementado en Prolog del método de las pertubaciones para el caso de tres procesos. El árbol de perturbaciones se ha representado por una estructura de dos argumentos: el primero representa un *nodo* del árbol y el segundo la lista de pares *(perturbacion, árbol_generado)* correspondiente a las distintas perturbaciones posibles para el *nodo*.

```
/* ARBOL */
/* arbol(nodo, [(perturbacion, arbol),....]) */
```

Un *nodo* se representa como una lista de estructuras, cada una de la cuales representa una fila. Cada *fila* consiste en un *estado* y una lista de mensajes.

```
/* NODO  */
/* nodo([fila(estado,lista_de_mensajes),.....]) */
```

Finalmente, el grafo que se especifica el protocolo se representa en una lista de *arcos*, cada uno de los cuales es una estructura de cuatro arguemntos,

el estodo inicial en que se dá o recibe la perturbación, el estado al que se llega al enviar o recibir la perturbación, el valor de la perturbación y el procesador origen o de la perturbación.

```
/* GRAFO:lista de arcos */
/* [arco(estadoA, estadoB, perturbacion, origen/destino),...] */
```

El programa consiste en la generación de un árbol final en el que no se generan nodos que no hubiesen aparecido previamente, perturbando el árbol inicial (en el que los procesadores se encuentran en estdo inicial y sin perturbaciones pendientes en la lista de mensajes), con las distintas perturbaciones del grafo.

```
:- perturba(grafo, arbol_inicial, arbol_final).
```

El resultado ha sido la comprobación de que no existen situaciones de bloqueo en el protocolo, ni interacciones no ejecutable. Si aparecen estados ambiguos, ya que un proceso puede estar en un mismo estado mientras los otros ocupan distintos estados. También se han detectado recepciones no especificadas, debidas a mensajes cuya llegada no se consideraba posible en un determinado estado, como la recepción de un mensaje de trabajo cuando un procesador está activo.

Publicaciones

El contenido original de esta tesis aparece en los siguientes artículos:

- *A transputer-Based Architecture to Exploit the Restricted And-Parallelism of Prolog.* Ruz, J.J., Saenz, F., Araujo, L. Proc. of Mediterranean Electrotechnical Conference (IEEE). Ljubliana, Slovenia, Yugoslavia, 22-24 May 1991, pp 1081-1084.
- *Explotación del Paralelismo "Y" Restringido sobre Arquitecturas con Memoria Distribuida* L. Araujo y J.J. Ruz. PRODE'91 2-4 Octubre 1991, pp 521-534.
- *Paralelismo Disyuntivo de Prolog sobre una Arquitectura con Memoria Distribuida* L. Araujo y J.J. Ruz. PRODE'92. Madrid.
- *Or-Parallel Execution of Prolog on a Transputer-Based System.* L. Araujo, J. Ruz. Transputer and Occam Research: New Directions. IOS Press, pp. 167-182.
- *A Transputer-based Prolog Distributed Processor.* L. Araujo, J. Ruz. Progress in Transputer and Occam Research. IOS Press, pp. 205-219.
- *PDP: Prolog Distributed Processor for Independent_AND/OR Parallel Execution of Prolog.* L. Araujo, J. Ruz. ICLP'94.

Referencias

[1] Ait-Kaci, H. *The WAM: A (Real) Tutotial.*

[2] Ali, K. A. M. and Roland Karlsson. *The Muse Approach to Or-Parallel Prolog. International Journal of Parallel Programming* Vol. 19 No. 2 April 1990.

[3] Ali, K. A. M. *OR Parallel Execution of Horn Clause Programs Based on WAM and Shared Control Information.* Technical report, Logic Programming System, SICS, Noviembre 1986.

[4] Ali, K. A. M. *OR Parallel Execution of Prolog on a Multi-Sequential Machine.* Accepted for Publication in IJPP, November 1986.

[5] Ali, K. A. M., Karlsson, R. *A Novel Method for Parallel Implementation of findall.* Proc. 6th ICLP. Pp. 235-245

[6] Ali, K.A.M. *A Method for Implementing Cut in Parallel Execution of Prolog.* Research Report SICS R87001 (1987).

[7] Ali, K.A.M., Karlsson, R. *Scheduling Or-Parallelism in Muse* Poc of the 8th ICLP. Paras. June, pp. 807-821.

[8] Apt, K. *Introduction to Logic Programming*

[9] Araujo, L. and Ruz, J.J. *Explotación del paralelismo "Y" Restringido de Prolog sobre arquitecturas con memoria distribuida.* Proc Prode'91, Malaga, Octubre (1991) 521-534.

[10] Araujo, L. and Ruz, J.J. *Paralelismo Disyuntivo de Prolog sobre una arquitectura con memoria distribuida.* Proc Prode'92, Madrid, Septiembre (1992).

[11] Araujo, L., Ruz. J. *Or-Parallel Execution of Prolog on a Transputer-Based System.* Transputer and Occam Research: New Directions. IOS Press, pp. 167-182.

[12] Araujo, L., Ruz. J. *A Transputer-based Prolog Distributed Processor.* Progress in Transputer and Occam Research. IOS Press, pp. 205-219.

[13] Araujo, L., Ruz. J. *PDP: Prolog Distributed Processor for Independent_AND/OR Parallel Execution of Prolog.* ICLP'94.

[14] Beer J., *Concepts, Design and Performance Analysis of a Parallel Prolog Machine* Springer-Verlag (1989).

[15] Biswas P., Su S., Yun D.*A Scalable Abstract Machine Model to Support Limited-OR(LOR)/Restricted-AND Parallelism(RAP) in Logic Programs.*

[16] Boyer, R.S., Moore, J.S. *The sharing of structure in theorem proving programs.* Machine Intelligence 7, 1972. Edinburgh University Press.

[17] Calderwood A., Szeredi, P. *Scheduling Or-parallelism in Aurora - the Manchester scheduler* 6th Conference on Logic Programming(1989) 419-435.

[18] Chang, J. -H., Despain, A. and Degroot, D., *AND-parallelism of logic programs based on a static dependency analisys*, Proc. Spring Compcon, IEEE, (1985), 218-225.

[19] Ciepielewski,A., Hausman, B., Haridi, S. *Initial Evaluation of a Virtual Machine for OR-Parallel Execution of Logic Programs* Fifth Generation Computer Architectures North-Holland 1986.

[20] Clark, K., Gregory, S. *PARLOG: Parallel Programming in Logic* Proc. of the Int. Conf. on Fifth Generation Computer System (1996), 299-306.

[21] Clocksin W. F., Alshawi H. *A Method for Efficiently Horn Clause Programs Using Multiple Processors* New Generation Computing,5 (1988) 361-376.

[22] Codognet, P. and Sola, T. *Extending the WAM for Intelligent Backtracking.* Proc. 8th ICLP, Paris, Junio (1991) 127-141.

[23] Conery, J. S.,*Parallel Execution of Logic Programs.* Kluwer Academic Publisher (1987).

[24] Conery, J. S., *Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors* International Journal of Parallel Programming 17(2), April (1988) 125-152.

[25] Cohen, J., Katcoff, J. *Symbolic Solution of Finite-Difference Equations* ACM transactions on Mathematical Software 3, 3 September 1977, pp. 261-271.

[26] Colmerauer, A. et al. *Un systeme de Communication Homme-Machine en Francais* Groupe de Recherche en Intelligence Artificelle, Universit d'Aix Marseille, 1973.

[27] Debray, S.K., Lin, N.-W., Hermenegildo, H. *Task Granularity Analysis.* SIGPLAN'90 Conf on Programming Language Design and Implementation, June 1990, pp 174-188.

[28] Debray, S.K., Lin, N. *Cost Analysis of Logic Programs* Proc of 8th ICLP. Paris, June 1991.

[29] Debray, S.K., Lin, N. *CASLOG (Complexity Analysis System for LOGig.* User manual. 1992.

[30] Degroot, D., *Restricted AND-Parallelism.* Proceeding of Int. Conf. Fifth Gen. Comp. Sys., ICOT (1984), 471-478

[31] Fagin, B. S. *A Parallel Execution Model for Prolog*, PhD Thesis, Technical report, University of California, Berkeley, California (1987).

[32] G. Gupta, M. Hermenegildo. *ACE:And/Or-parallel Coying-based Execution of Logic Programs.* 1991ICLP Workshop on Parallel Logic Programming, Springer Verlag, LNCS.

[33] Green, C.C. *Theorem proving by resolution as a basis for question-answering Systems.* Machine Intelligence 4, 1969. Edinburgh University Press, pp 183-205.

[34] Hausman, B. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, SICS (1990).

[35] Hermenegildo, M., *An abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Program in Parallel.* PhD thesis, U. of Texas at Austin (1986).

[36] Hermenegildo, M., *Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs.* Proceedings of the Fourth International Conference on Logic Programming, MIT Press, 1987, pp. 556-576.

[37] Hermenegildo, M. and Greene, K., *The &-Prolog system: Exploting Independent And-Parallelism Transparently.* Technical report, Association for Logic Programming, June 1990.

[38] Kaksuk, P. Wise, M.J. *Implementations of Distributed Prolog* Wiley 1992.

[39] Karjoth, G., Sjodin, P., Weckner, S. *A Sophisticated Environment for Protocol Simulation and Testing.* SICS R86004. Research Report.

[40] kergommeaux J.C., Robert, P. *An Abstract Machine to Implement Or-And Parallel Prolog Efficiently.* J. L. P. 1990, 8 Pp. 249-246.

[41] Kergommeaux, J.C., Codognet, P. *Parallel Logic Programming Systems* RR-891-I, May, 1992.

[42] Kergommeaux, J.C., Baron, U.C., Ratcliffe, M. *Performance Analysis of a Parallel Prolog: A Correlated Approach* Proc. PARLE'89, Eindhoven, The Netherland.

[43] Kowalski, R. *Predicate Logic as a Programming Languaje.* Proc. IFIP 74, North-Holland, 1974, pp. 569-574.

[44] Kuchen, H., Wagener, A. *Comparison of Dynamic Load Balancing Strategies* Tech. Report 90-5 . Aachener Informatik-Berichte.

[45] Jelly, I.E., Gray, J.P. *Prototyping Parallel Systems: A Performance Evaluation Approach* Proc. Fifth Int. Conf on Parallel and Distributed Computing and Systems, October 1992.

[46] Jelly, I.E., Morris, S.A. *Mixed Language Programming for Transputer Networks.* Transputer and Occam Research: New Directions. IOS Press, pp. 41-54.

[47] Kale, L. V. *The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs* Proc ICLP Melbourne, Australia (1987) 616-632.

[48] Kernighan, B.W., Ritchie, D.M. *El lenguaje de programación C.* Prentice Hall.

[49] Ivie, J. *Some MACSYMA Programs for Solving Recurrence Relations* ACM transactions on Mathematical Software 4, 1 March 1978, pp. 24-33.

[50] Lin, Y.-J. and Kumar, V., *AND-parallel execution of Logic Programs on a Shared-Memory Multiprocessor.* The Journal of Logic Programming, 1991:10:155-178 (1991).

[51] Lloyd, J. W. *Foundations of Logic Programming* Springer-Verlag, 1984.

[52] Lopez Garcia, P., Hermenegildo, M. *Toward Dynamic Term Size Computation via Program Transformation* Proc. PRODE'93. Blanes, pp. 73-91.

[53] Masuzawa, H. et al. *Kabu wake parallel inference mechanism and its evaluation* FJCC (1986) 955-962.

[54] Moreno-Navarro, J.J., Rodriguez-Artalejo, M. *Logic Programming with Functions and Predicates: The Languaje BABEL*, J. Logic Programming, 12, 1992, 189-223.

[55] Orava, F. *Verifying Safety and Deadlock Properties of Networks of Asynchronously Communicating Processes.* SICS R88020. Research Report.

[56] Pehrson, B. *Verification of Protocols.* SICS R88012. Research Report.

[57] Robinson, J.A. *A machine-oriented logic based on the resolution principle.* Journal of the Association for Computer Machinery 12, 1965, pp. 23-41.

[58] Reed, D.A., Fujimoto, R.M. *Multicomputer Networks: Message-Based Parallel Processing.* MIT Press, 1987.

[59] Ruz, J.J., Saenz, F., Araujo, L. *And-parallel Execution of Prolog on a Distributed Architecture.* Proc. ISMM International Symposium 1990. Lugano Switzerland. June 19-21, 1990

[60] Ruz, J.J., Saenz, F., Araujo, L. *A transputer-Based Architecture to Exploit the Restricted And-Parallelism of Prolog.* Proc. of Mediterranean Electrotechnical Conference (IEEE). Ljubliana, Slovenia, Yugoslavia, 22-24 May 1991, pp 1081-1084.

[61] Shapiro, E. *Concurrent Prolog* MIT Press (1988).

[62] K. Shen and D. H. D. Warren. *A simulation study of the Argonne model for Or-parallel execution of Prolog.* SLP, 1987.

[63] Shen K, Hermenegildo M. *A Simulation study of Or- and Independent And-parallelism.*

[64] Sugie, M. Yoneyama, M., Tarui, T. *Load-Dispatching Strategy on Parallel Inference Machine* Proc. Int. Conf. on Fifth Generation Computer System 1988, pp. 987-993.

[65] Tanenbaum, A.S! *Computer Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[66] Tick, E. *Compile-Time Granularity Analysis for Parallel Logic Programming Languages* New Generation Computing, 7 (1990) 325-337.

[67] Tseng, C., Biswas, P. *A Data-Driven Parallel Execution Model for Logic Programs* Proc ICLP Seattle (1988) 1204-1222.

[68] Tubella, J., Gonzalez, A. *Measuring Scheduling Policies in Pure OR-Parallel Programs* PRODE'93 Sep. 1993, Blanes, pp. 57-71.

[69] Ueda, K. *Guarded Horn Clauses: A Parallel Logic Programming Languaje with the Concept of a Gard.* Technical Report TR-208. ICOT, Tokyo, Japan (1986).

[70] Van Roi, P. *Can Logic Programming Execute as Fast as Imperative Programming* PhD thesis. U. of California (1990).

[71] Verden, A. and Glaser, H., *Independent AND-Parallel Prolog for Distributed Memory Architectures.* Technical Report CSTR 90-17. University of Southampton.

[72] Warren, D.H.D., *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, (1983).

[73] Warren. D.H.D., *OrParallel Execution Models of Prolog.* DATSOFT'87.

[74] D.H.D. Warren. *The andorra principle.* Internal report, Gigalips Group, 1988.

[75] Warren. D.H.D., *The SRI Model for Or-Parallel Execution of Prolog-Abstract Design and Implementation Issues* 4th symposium on Logic Programming, (1987) 46-53.

[76] Westphal, H, Robert, P, Chassin, J, Syre, J. *The PEPsys model: Combining Backtracking, AND- and OR-parallelism.* SLP 1987.

[77] Yasuhara, H., Nitadori, K. *ORBIT: A Parallel Computing Model for Prolog.* New Generation Computing, 2(1984) 277-288.

[78] Yuan, S.M. *An Efficient Periodically Exchanged Dynamic Load-Balancing Algorithm.* Int. Journal of Mini and MicroComputers. 1, 1990, pp. 1-7.

[79] Zafiropulo, P., West, C., Rudin, H., Cowam, D., Brand, D. *Towards Analizing and Synthesizing Protocols.* IEEE Transactions on Communications, Vol Com-28, NO. 4. April 1980.

[80] Zhong, X., Tick, E., Duvvuru, S., Hansen, L., Sastry, A.V.S. and Sundararajan, R. *Towards an Efficient Compile-Time Granularity Analysis Algorithm.* Technical report CIS-TR-91-19, Department of Computer and Information Science. University of Oregon.

[81] *ANSI C Toolset User Manual*, Inmos Ltd, 1988