UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

Implementación del no determinismo y optimización del uso de memoria en la ejecución paralela conjuntiva de programas lógicos

Autor: Manuel Carro Liñares

Tutor: Manuel V. Hermenegildo Salinas

Agradecimientos

Quiero agradecer, ante todo, el apoyo que mi tutor me ha prestado siempre, no sólo en la elaboración de este trabajo, sino en todas las tareas llevadas a cabo en nuestro laboratorio: por dar siempre el consejo acertado, por no desmayarse ante los contratiempos y por tener siempre un minuto disponible. Quiero también agradecer a mis compañeros de laboratorio el que sepan soportar mis (malas) bromas y el que creen un ambiente en el que trabajar es realmente agradable. Estoy especialmente en deuda con Paco y María por haber leído y corregido borradores de este trabajo.

Quisiera agradecer también las apreciaciones, opiniones e intercambio de ideas que he mantenido con varios compañeros, especialmente Kish Shen, de la Universidad de Bristol y Gopal Gupta, Enrico Pontelli y Donxing Tang, de la Universidad del Estado de Nuevo México en Las Cruces.

A mis padres les debo el apoyo incondicional a lo largo de mi vida y la comprensión con la que siempre he contado, que nunca podré devolver en su justa medida.

Y a Verónica, con su ejemplo de coraje y ánimo, le debo darme fuerzas y motivos para seguir trabajando.

Manuel Carro Liñares

Universidad Politécnica de Madrid

Facultad de Informática

Índice general

Resumen

Los lenguajes lógicos (y declarativos en general) están teniendo cada vez más auge como medio para escribir sistemas complejos (sistemas expertos, programas de Inteligencia Artificial, etc.), pero adolecen de una menor eficiencia en términos de velocidad. Una de las formas de paliar esta deficiencia es usar procesamiento paralelo.

Existen diferentes propuestas para la ejecución paralela de lenguajes lógicos, de los cuales tomaremos a Prolog como ejemplo de trabajo. Un comportamiento generalizado entre las propuestas es que las más sencillas son a la vez las más lentas y ávidas de memoria. Por ello henos optado por partir de lo que hoy por hoy es el diseño más eficiente para la ejecución de Prolog y trabajar sobre una adaptación de ese diseño para la ejecución en paralelo, estudiando los problemas y necesidades para conseguir una ejecución correcta y eficiente, sobre todo en el sentido del uso de memoria, pero sin perder de vista la velocidad del sistema.

Uno requisito deseable es preservar la semántica del lenguaje secuencial en la ejecución paralela. Los problemas principales a los que nos enfrentamos vienen dados por la necesidad de implementar el *backtracking* en un sistema con varios procesadores basándose en un sistema plenamente ideado con vistas a la ejecución secuencial. La comunicación, sincronización entre procesadores y recuperación de memoria son los principales puntos que hemos tratado de resolver en este trabajo. Las soluciones propuestas se han desarrollado al nivel de comunicación entre tareas secuenciales, abstrayendo la maquinaria necesaria para la ejecución secuencial. El propósito de ello es no utilizar en exceso características propias de una implementación particular con objeto de facilitar adaptación a diferentes sistemas. Se ha realizado como parte del trabajo una implementación parcial de las técnicas descritas y se ha llevado a

cabo una evaluación de su eficiencia en términos de velocidad y consumo de memoria. Finalmente se propone una optimización para mejorar el consumo de memoria y la velocidad en la ejecución de programas con componente determinista.

Capítulo 1

Introducción

1.1. El procesamiento paralelo

El uso de ordenadores como herramientas de cálculo ha permitido realizar grandes avances en determinados campos técnicos. Pero, aparte del diseño de nuevos y más completos lenguajes como parte de la evolución de la Ciencia de la Informática, la creciente complejidad de los programas (como, por ejemplo, sistemas expertos, bases de datos avanzadas, sistemas de lenguaje natural y analizadores y compiladores de muy alto nivel) ha llevado a la necesidad de realizar *software* con un nivel de abstracción más cercano al pensamiento humano. Con el propósito de facilitar la tarea del programador se han empezado a utilizar de forma cada vez más extendida varias familias de lenguajes antes consideradas rarezas de laboratorio, entre las cuales cabe reseñar los lenguajes funcionales y lógicos.

El avance en el nivel de abstracción de dichos lenguajes se ha pagado con un retroceso en una de las cualidades más atrayentes de los ordenadores: la velocidad de ejecución. Hay varios métodos (no excluyentes) para paliar este retroceso: realizar una compilación más inteligente del lenguaje, de manera que se explote al máximo la capacidad del ordenador, diseñar procesadores cada vez más rápidos y emplear máquinas paralelas capaces de efectuar varias tareas a la vez. Todos ellos tienen limitaciones: una vez efectuada la compilación más eficiente existente no será posible realizar una ejecución más rápida; una vez diseñado el procesador más rápido posible, las limitaciones físicas de la materia impedirán mayor velocidad; una vez explotado

todo el paralelismo en un problema dado, el añadir más procesadores no acelerará la ejecución.

Tenemos varias razónes que nos mueven a considerar el paralelismo como campo de estudio. Por una parte la ejecución paralela de lenguajes simbólicos es un campo relativamente joven, en el que aún queda bastante por hacer. Por otra parte la aparición de máquinas paralelas asequibles abre un campo práctico a los sitemas paralelos. Y, por último, el paralelismo es, en principio, escalable con el tamaño del problema a resolver. Ello significa que un mismo programa resolviendo un problema mayor podría (teóricamente) mantener la misma velocidad si se ejecuta en una máquina con más procesadores, manteniendo la misma técnica de compilación y la misma velocidad por procesador. Lo único que necesitamos es *software* diseñado para ser ejecutado en paralelo.

Existen al menos dos formas de obtener un *software* de este tipo: mediante la indicación explícita por parte del programador del posible paralelismo existente en el programa y de cómo se ha de explotar, gracias a la utilización de lenguajes que incluyan construcciones para expresar paralelismo, o mediante el uso de compiladores que realicen una paralelización automática del programa.

La primera técnica requiere que sea el programador el encargado de determinar las dependencias entre las diferentes partes del programa, su secuenciamiento y sincronización. Esta tarea, delicada y sujeta a múltiples errores, incrementa seriamente la dificultad de un trabajo ya considerablemente complejo: el diseño, escritura y depuración de programas. La segunda técnica, en cambio, permite aprovechar el rendimiento del multiprocesador sin que el programador tenga que preocuparse de la realización de estas tareas. En particular, el programador se ve no sólo aliviado de tener que realizar por si mismo el análisis del algoritmo con vistas a su paralelización, sino que se ve además liberado de tener en cuenta las características propias de la

máquina en que se ejecutará su programa. Por supuesto, ésto no quiere decir que no se deba permitir al programador expresar explícitamente el paralelismo cuando lo desee, y elaborar programas destinados a ejecutarse en una máquina dada.

El diseño de un sistema tal tiene dos vertientes independientes: por una parte la creación de un compilador capaz de analizar programas secuenciales y transformarlos en programas en los cuales el paralelismo aparezca explícitamente, y por otra la creación de sistemas de ejecución capaces de ejecutarlos en paralelo. Ninguna de esas tareas es fácil.

La escritura del compilador se ve simplificada cuando se toman en consideración los lenguajes declarativos. Estos lenguajes, y en particular los llamados lenguajes lógicos, especifican únicamente (al menos en principio) *qué* se debe hacer para resolver el problema, sin imponer ningún orden en particular, siendo así mucho menos complejo automatizar el proceso de paralelismo. Es importante destacar además que, gracias a los fundamentos matemáticos y declarativos de estos lenguajes, es posible establecer una base firme sobre la que fundamentar los criterios de correción de las técnicas de paralelización utilizadas. Sin embargo, sería un error pensar que en este caso la paralelización es un proceso sencillo. Por el contrario, incluso para los lenguajes declarativos, es una tarea compleja, con la que no siempre se llega a aprovechar todo el paralelismo y en la que el estilo de programación utilizado al desarrollar el programa que se desea paralelizar es muy importante.

Por otra parte, y debido al alto nivel de los lenguajes declarativos, el *run–time system* se complica. Sólo en una fecha tan reciente como 1983 se pudo disponer de la primera descripción completa (y críptica) de una máquina abstracta para la ejecución secuencial de Prolog [War83]. Ya el funcionamiento de esta máquina abstracta es complejo en el caso secuencial; cuando deseamos que una implementación de eficiencia comparable se ejecute en paralelo, los problemas se multiplican. Dos son los criterios

fundamentales, desde el punto de vista del usuario, que deben satisfacerse al realizar una implementación paralela:

- Preservar la semántica del lenguaje. Deseamos que la ejecución de los programas ya escritos siga obteniendo los mismos resultados. Si el nuevo lenguaje permite (característica deseable) hacer explícitas qué partes deben ejecutarse en paralelo, las nuevas construcciones que lo hacen posible deberían asemejarse lo más posible a las del lenguaje inicial, y no cambiar demasiado su semántica. El objetivo de esto es que el usuario se sienta cómodo con el nuevo lenguaje si decide escribir directamente programas paralelos.

- Asegurar que la paralelización no va a retardar la ejecución del programa. El propósito último y único del paralelismo es velocidad, porque no hay nada que una máquina con un numero finito de procesadores pueda hacer que no pueda hacerse en un único procesador, excepto ir más deprisa. Puede parecer extraño que el ejecutar un programa en varios varios procesadores en vez de en uno pueda retardar un programa, pero todo depende del método. Es posible que la carga añadida al sincronizar y distribuir el trabajo entre varios procesadores sobrepase la ganancia obtenida con el proceso en paralelo.

Vamos a revisar brevemente dónde puede explotarse el paralelismo en los lenguajes lógicos, y apuntar los problemas con que nos podemos encontrar a la hora de realizar una implementación.

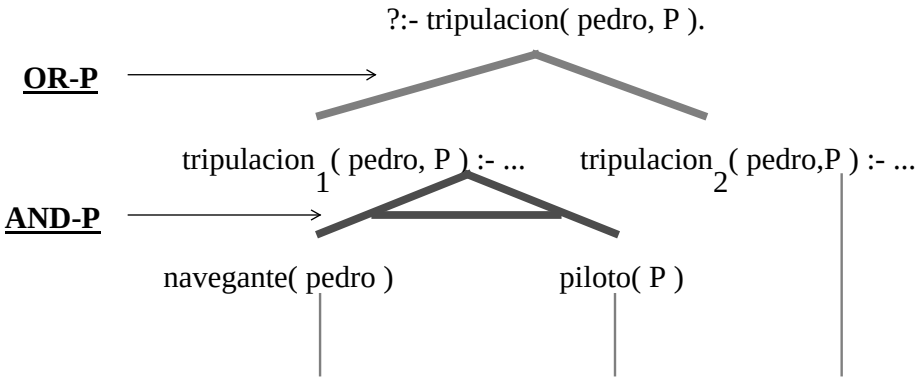

Figura 1.1: Paralelismo Or– y And– en programas lógicos

1.2. Tipos de paralelismo en programas lógicos

El paralelismo existente en un programa lógico puede ser, básicamente, de dos tipos [Con87]: **paralelismo–And** y **paralelismo–Or**.[1] Considérese el siguiente ejemplo para determinar la tripulación de un avión (figura 1.1):

```
tripulacion(X, Y) :- navegante(X), piloto(Y).
tripulacion(X, Y) :- mecanico(X), piloto(Y).
```

El significado de la primera cláusula es el siguiente: una tripulación para un avión puede formarse con `X` e `Y`, si `X` es un `navegante` e `Y` es un `piloto`. La única diferencia existente entre ésta y la segunda cláusula es que en la segunda, `X` debe ser un `mecanico` en vez de un navegante.

Como se puede ver en la figura 1.1, existen dos caminos alternativos que satisfacen la consulta:

[1] Existe también el paralalelismo en la unificación [Bar90], pero no lo trataremos aquí.

```
:- tripulacion(pedro,P).
```

los cuales se corresponden con las dos cláusulas que definen `tripulacion`. El paralelismo resultante de la ejecución simultánea de cada una de los posibles caminos realizada por procesadores diferentes se denomina paralelismo–Or. El paralelismo–Or se corresponde con la ejecución paralela de las diferentes ramas alternativas del árbol de prueba.

Si consideramos la ejecución de una de las cláusulas alternativas, por ejemplo la primera, vemos que para satisfacer `tripulacion` sería necesario satisfacer tanto `navegante(pedro)` como `piloto(P)`. El paralelismo resultante de la ejecución simultánea de ambas metas realizada por procesadores diferentes se denomina paralelismo–And.

El paralelismo–Or es característico de problemas de búsqueda. Conceptualmente este tipo de paralelismo es bastante sencillo, gracias a lo cual su implementación está prácticamente resuelta [War87b, Lus88, Kal87, Hau90, Sze89, War87a, CH83, Kar92, Car90]. Esto es debido a que (en programación lógica pura) no existe ningún tipo de dependencia entre las ramas alternativas, y pueden explorarse de manera simultánea sin (apenas) comunicación.

El paralelismo–And se presenta tanto en problemas determinísticos como no determinísticos, lo que amplía considerablemente su campo de acción. Sin embargo su implementación es una tarea particularmente delicada debido a las interrelaciones entre las metas que se desean ejecutar paralelamente.

1.3. El paralelismo–And y las dependencias

Ciertos programas con paralelismo–And pueden ejecutarse en paralelo en un tiempo mayor que en una ejecución secuencial [HR95], aún suponiendo que cada pro-

cesador ejecuta su parte tan rápidamente como si fuera secuencial y que no se emplea ningún tiempo para efectuar la sincronización de los procesadores. Supongamos que se realiza una implementación en la que, durante la ejecución de un programa al invocar una cláusula, se permite que todas las metas en el cuerpo de dicha cláusula se ejecuten simultáneamente sin ningún tipo de restricción. Consideremos la siguiente cláusula:

```
piloto(P) :- licencia(P), medico(P).
```

cuyo significado es el siguiente: `P` es un `piloto` si `P` tiene una `licencia` y ha pasado con éxito el examen `medico`. Supongamos que la consulta inicial es `:- piloto(pedro)`. La ejecución secuencial ejecutaría primero la meta `licencia(pedro)` y a continuación, si tiene éxito, ejecutaría la meta `medico(pedro)`. La ejecución paralela en la implementación propuesta ejecutaría las metas `licencia(pedro)` y `medico(pedro)` en paralelo. Es evidente que si la ejecución secuencial tuvo éxito, la ejecución paralela también lo tendrá y en menor tiempo.

Supongamos, en cambio, que la consulta es `:- piloto(P)`. La ejecución secuencial ejecutaría primero la meta `licencia(P)`. Supongamos que la variable `P` se unifica con la constante `pedro`. A continuación se ejecutaría la meta `medico(pedro)`, lo que como vimos tenía éxito. Sin embargo, la ejecución en paralelo de las metas `licencia(P)` y `medico(P)` puede dar lugar a resultados inconsistentes si el orden de los hechos hace que el procesador que ejecute `licencia(P)` unifique la variable `P` a `pedro` y el procesador que ejecute `medico(P)` unifique `P` a cualquier otro individuo. Al ver que las ligaduras son inconsistentes habría que reejecutar las metas hasta que no lo fueran, pudiéndose así superar el tiempo de ejecución secuencial de forma considerable. La razón de ello es que en la ejecución secuencial la primera meta restringe el campo de búsqueda de la segunda mediante la instanciación de la variable compartida, lo que no ocurre en el caso de la ejecución paralela.

El problema, sin embargo, no se presenta únicamente cuando la misma variable aparece en dos metas que se van a ejecutar en paralelo. Considérese la segunda cláusula del ejemplo `tripulacion`:

```
tripulacion(X,Y) :- mecanico(X), piloto(Y).
```

Inicialmente, podría parecer que los problemas habidos en el anterior ejemplo no aparecerían aquí. Sin embargo, bastaría una consulta del tipo

```
:- tripulacion(Z,Z).
```

es decir, aquella en la que se desea que sea una única persona la que constituya la tripulación, para provocar idénticos problemas. Vemos así que los conflictos en las instanciaciones de las variables se deben a la ejecución en paralelo de metas que comparten alguna variable en tiempo de ejecución, incluso cuando en el texto de la cláusula no aparecen variables compartidas.

Un método para resolver el problema de las dependencias sería generar todas las posibles soluciones para cada meta y, posteriormente, eliminar las inconsistencias, determinando así el conjunto de soluciones para la cláusula [RK89]. La principal ventaja de este método es la sencillez de su implementación. Su desventaja fundamental es la ineficiencia, provocada tanto por la gran cantidad de trabajo redundante realizado, como por el alto coste debido a la función que elimina las inconsistencias y por la gran cantidad de memoria necesaria para el almacenamiento de datos intermedios.

Un método mucho más práctico, y de hecho el utilizado en la mayoría de los diferentes modelos de paralelismo–And existentes, consiste en determinar las dependencias entre las metas del cuerpo de la cláusula. Una vez determinadas dichas dependencias, cada modelo de paralelismo–And se diferencia en el efecto que dichas

dependencias tienen sobre el número de metas a ejecutar en paralelo y la comunicación existente entre las mismas. Veamos algunas soluciones que han sido propuestas para establecer las dependencias entre las diferentes metas del cuerpo de una cláusula.

1.4. Métodos de inferencia de dependencias

Una de las primeras aportaciones a la solución de este problema fue dada por Conery [Con83]. Su método estaba basado en un algoritmo de ordenación capaz de determinar las dependencias entre metas en tiempo de ejecución. Sin embargo, este método disminuía significativamente el rendimiento del sistema debido a su alto coste computacional.

Posteriormente, Chang *et al.* [CDD85] propusieron otro sistema, con el cual se podían inferir dependencias entre los datos en tiempo de compilación, a partir de la información proporcionada por el programador. Esta característica (el ser realizado totalmente en tiempo de compilación), es su mayor ventaja: coste computacional nulo en tiempo de ejecución, y su mayor inconveniente: la información resultante se basaba en un análisis conservador, es decir, en caso de duda se escogería el caso más desfavorable para preservar la corrección del método. Ello provoca necesariamente una pérdida de cierta cantidad de paralelismo existente en el programa, pérdida que aunque en la actualidad, dado el alto grado de exactitud al que se ha llegado en el análisis, parece razonable, en su tiempo pareció excesiva.

El método propuesto por DeGroot en su esquema de *paralelismo–And restringido* (*Restricted And–Parallelism*, RAP) [DeG84] aportaba las ventajas de ambos métodos al establecer un compromiso entre la determinación de dependencias entre metas del cuerpo de una cláusula realizada totalmente en tiempo de ejecución y la realizada totalmente en tiempo de compilación. La solución fue generar en tiempo de

compilación varios grafos, uno por cada posible modo de activación de la cláusula, en vez de elegir siempre el peor caso. Estos grafos se combinaban produciendo una única *expresión de grafo de ejecución* (**E**xecution **G**raph **E**xpression, abreviado como **EGE**). Con ello se conseguía, por una parte, simplificar considerablemente el sistema necesario en tiempo de ejecución y, por otra, ampliar significativamente el número de casos en los que el paralelismo era detectado. Sin embargo, esta solución por una parte seguía adoleciendo de falta de precisión y expresividad a la hora de determinar las dependencias entre las metas, no aportaba una especificación del procedimiento a seguir en caso de fallo de alguna de las metas ejecutadas en paralelo, ni indicaba qué algoritmos o heurísticas eran adecuadas para generar las expresiones. Por otra, la gran complejidad de los tests propuestos limitaba su rendimiento.

1.5. Paralelismo-And independiente

La mayoría de las respuestas a estos problemas fueron aportadas por Hermenegildo [Her86a, Her86b, Her87, HN86] al desarrollar un conjunto de expresiones de grafo de ejecución mucho más sencillo y potente (denominadas *expresiones de grafo condicional*, **CGE**) y proporcionar un semántica operacional para cláusulas lógicas completa. Esta versión extendida del paralelismo–And restringido se denominó *paralelismo–And independiente*. Así mismo, el autor propuso una implementación basada en la Máquina Abstracta de Warren (WAM) [War83], siendo el suyo el primer esquema en el que se proporcionaba una descripción de los métodos de implementación. Este tipo de paralelismo–And se denomina *independiente* debido a su principal característica: una vez establecidas las dependencias entre variables, únicamente se permite ejecutar en paralelo aquellas metas que no compartan variables en tiempo de ejecución, de forma que sus resultados no sean inconsistentes entre sí [HR89].

Este método asegura que la ejecución paralela se puede realizar en el mismo o menor tiempo que la ejecución secuencial al preservar la complejidad esperada por el programador, pero aumentando el rendimiento gracias a la ejecución en paralelo del mayor número de metas posible. La ausencia de variables compartidas hace que la sincronización entre los procesadores únicamente tenga que referirse al control, ya que no habrá competencia por las ligaduras de variables. Esto simplifica grandemente el diseño y la implementación del modelo de ejecución.

1.6. Paralelismo-And dependiente

Existe otro tipo de paralelismo–And en el que, una vez determinadas las dependencias entre variables, sí se permite la ejecución paralela de las metas dependientes. Los problemas de ligaduras se resuelven mediante la comunicación entre las diferentes metas dependientes, siendo el canal de comunicación precisamente, la variable que da lugar a su dependencia. Este método ha dado lugar al denominado paralelismo–And *stream*.

La principal desventaja del paralelismo–And stream radica en la gran complejidad existente a la hora de implementar su *backtracking*, es decir, en la complejidad de implementar mecanismos de retroceso en presencia de procesos no deterministas. Como resultado, en muchos de los sistemas que utilizan este tipo de paralelismo se ha optado por abandonar el backtracking y, por tanto, las búsquedas no deterministas del tipo *"don't know"*, es decir, aquellas que permiten encontrar todas las soluciones a un determinado problema. En cambio, se ha adoptado el indeterminismo del tipo *"don't care"* [2] en el que, en el momento en que se escoge una cláusula[3], se sigue

[2]Traducido en ocasiones por: indeterminismo *"no sé"*, indeterminismo *"no importa"*.

[3]Habitualmente una cláusula se considera escogida cuando, tras haberse unificado su cabeza satisfactoriamente, se pasan con éxito una serie de test sencillos denominados *guardas*.

adelante sin poder ya volver al punto pasado, perdiendo así una de las características más interesantes y útiles de los programas lógicos. PARLOG [CG83], Concurrent Prolog [Sha83], y GHC [Ued86] son ejemplos de este tipo de lenguajes denominados de *elección irrevocable* ("committed-choice").

El paralelismo–And independiente, en cambio, no sólo mantiene el indeterminismo "don't know" al permitir el backtracking, sino que mantiene totalmente la semántica operacional de Prolog. Es decir, la ejecución de un programa en un sistema basado en este tipo de paralelismo debe producir el mismo efecto que su ejecución en un sistema secuencial, siendo la única diferencia la mejora obtenida en el rendimiento del programa.

Entre las propuestas de paralelismo–And dependiente con *backtracking* se encuentra el esquema DDAS (Dynamic Dependent And–Parallel Scheme), que retiene la semántica de Prolog pero permitiendo la ejecución paralela de objetivos que compartan variables [She92b, She92a]. El rango de aplicaciones en las que DDAS puede extraer paralelismo es potencialmente mayor que con el paralelismo–And independiente, pero la gran complejidad de su implementación hace que sea inherentemente mas lento incluso en las partes puramente secuenciales. Sin embargo es una de las propuestas más sólidas hacia el paralelismo–And dependiente.

Existen otras alternativas que, aunque no se tratarán en este trabajo, conviene mencionar. Una de ellas es mantener la semántica "don't know" permitiendo únicamente el paralelismo-And stream cuando los objetivos son deterministas. Esta solución, utilizada en el modelo Andorra–I [SCWY91b, SCWY91a] restringe la cantidad de paralelismo. La semántica dada por el Principio Andorra a los programas basados en cláusulas de Horn no es la misma que la dada por Prolog, de forma que los programas Prolog tienen que pasar por un complicado preproceso para ser ejecutados en Andorra–I con los mismos resultados. Recientemente se ha propuesto

una solución que combina las ventajas de éste método con las del paralelismo–And independiente: el modelo IDIOM [GSCYH91]. Finalmente, es importante señalar los diferentes modelos propuestos que combinan el paralelismo–And con el paralelismo–Or [GJ89, GHPSC92].

1.7. Objetivos

Como hemos mencionado, en el modelo de paralelismo–And independiente un conjunto de metas en el cuerpo de una cláusula podrá ser ejecutado en paralelo si se demuestra que no comparten variables en tiempo de ejecución. No entraremos en este trabajo a examinar como se paraleliza un programa, ni desde el punto de vista de un programador que lo hace a mano ni desde el punto de vista de la persona que escribe un compilador paralelizante. Supondremos que los programas ya han sido correctamente paralelizados, de manera que ejecutaremos objetivos independientes. Nos concentraremos en el estudio de la ejecución a nivel de tarea paralela, e intentaremos evaluar la ganancia en términos de velocidad y su coste en términos de gasto de memoria. En particular veremos:

- Una somera introducción a la Warren's Abstract Machine (WAM) y a las modificaciones propuestas por M. Hermenegildo (RAP–WAM, PWAM). Nos concentraremos en las partes que resultan más interesantes para nuestros propósitos.
- Problemas que aparecen en relación al *backtracking* y soluciones propuestas.
- Un estudio de un esquema paralelo de *backtracking* y una propuesta de la maquinaria necesaria para su implementación. El nivel de esta exposición no bajará hasta su codificación interna, pues en este nivel se depende de características que varían mucho de un sistema a otro. El propósito es estudiar

problemas y diseñar soluciones que dependan del esquema de ejecución y no de la implementación en particular.

- Datos cuantitativos obtenidos mediante una implementación paralela (parcial) del esquema anteriormente discutido. Estos datos son, por supuesto, dependientes de la implementación, pero sirven como guía para estimar cuanto se puede ganar y a qué precio. Hay que hacer notar que la implementación no esta optimizada en ningún aspecto (ni siquiera adoptando el modelo de ejecución más favorable), de forma que las cifras aquí presentadas son una cota pesimista de la eficiencia esperada.

- Propuesta de un esquema destinado a reducir el consumo de memoria y tiempo de ejecución en la ejecución paralela de cierta clase de objetivos. Este esquema intenta acercar lo más posible la ejecución de programas deterministas paralelos a los secuenciales, y aprovechar información sobre este determinismo para realizar un planificación más favorable.

Capítulo 2

&–Prolog: semántica y arquitectura

En este capítulo repasaremos brevemente los aspectos más relevantes del modelo de ejecución de Prolog y la arquitectura de la WAM, como paso previo a la descripción del lenguaje &–Prolog y de una arquitectura para su ejecución basada en la WAM, la PWAM [HG91, Her86b].

2.1. Prolog en breve

Prolog es un lenguaje basado en la lógica de primer orden. Un programa Prolog es un conjunto de predicados en forma de cláusulas de Horn [Kow74]. La definición de cada predicado puede verse como una serie de reglas que ofrecen alternativas para resolver un problema. Cada una de las cláusulas contiene una descripción de pasos sucesivos que hay que dar para resolver ese problema.

Las características más distintivas de Prolog como lenguaje de programación son el uso de variables lógicas (objetos que pueden manejarse y pasarse como argumentos antes de tener un valor definido) y el *backtracking* incluído en el lenguaje. El *backtracking* es un modo de simular en una máquina determinista el comportamiento de una máquina no determinista.

Operacionalmente Prolog efectúa una búsqueda en profundidad y de izquierda a derecha en un árbol And–Or. Esta búsqueda es más barata que una búsqueda en amplitud, pero adolece de falta de completitud. Uno de los problemas por los que pasa un principiante en Prolog (y los que ya no deberíamos ser tan principiantes) es

escribir programas que realizan computaciones infinitas al entrar repetidamente por la misma rama una y otra vez, cuando la solución esta en una rama diferente.

2.2. WAM

Las primeras implementaciones de Prolog fueron intérpretes. Un programa interpretado es habitualmente más lento que su equivalente ejecutado tras ser compilado, pues el proceso de compilación, mediante un análisis más o menos detallado del programa, permite eliminar el trabajo redundante o innecesario que un intérprete suele realizar.

La WAM es una definición de un lenguaje para la compilación de Prolog (denominado comúnmente "código WAM") y de una máquina abstracta (datos e instrucciones) para su ejecución. En la actualidad se usan diferentes variantes de la WAM para implementar Prolog, pero todas ellas comparten las mismas características esenciales. Hoy por hoy, con pocas excepciones la WAM es *de facto* la base de cualquier Prolog que pretenda ofrecer unas prestaciones competitivas. Habitualmente la WAM se simula, o se compila el código Prolog a código máquina que efectúa una ejecución similar a la WAM.

Lo que sigue es una somera descripción de la arquitectura de la WAM. El lector interesado en profundizar más en ella puede referirse a [AK91].

2.2.1. Datos

Los datos que tiene que representar la WAM son los mismos presentes en Prolog: variables libres, constantes y estructuras, asi como las unificaciones entre ellas.

La WAM supone celdas de memoria etiquetadas para distinguir los diferentes

tipos de datos. Las variables se representan como celdas con la etiqueta VAR y que apuntan a sí mismas. Las constantes se representan con celdas con la etiqueta CONS conteniendo una representación única de una constante. La correspondencia entre la constante que el programador ve o escribe en Prolog y la representación en el código WAM se determina en tiempo de compilación. Por último, las estructuras se representan como una celda con la etiqueta STR que contiene como valor una codificación del functor principal de la estructura (nombre + aridad) y a la que siguen n celdas (una por cada argumento de la estructura) que apuntan a la representación de los argumentos.

La unificación de una variable con otro objeto se realiza cambiando el puntero de la variable para que apunte al nuevo objeto. Esta descripción es muy simplista: en realidad hay una serie de reglas para determinar la dirección de los punteros en el caso de unificaciones entre variables, pero su explicación necesita un conocimiento más completo del funcionamiento de la WAM, y no es imprescindible para seguir este trabajo.

2.2.2. Memoria

La memoria de la WAM consta de varias partes, con papeles claramente definidos.

Código (Code area): las instrucciones WAM correspondientes a un programa Prolog.

Registros de argumentos (Argument registers): celdas etiquetadas destinadas a pasar datos a/desde los predicados Prolog.

Pila de unificación (Push–down list, PDL): pila temporal utilizada durante la unificación de estructuras.

Montón (Heap): zona de memoria en la cual se almacenan variables, constantes, y estructuras durante la ejecución.

Pila local (Local stack, environment stack): almacena variables locales a una clausula y punteros a la continuacion de una clausula. Es, en cierto modo, similar a la pila usada para almacenar entornos en un lenguaje de programación imperativo o funcional, pero la necesidad de tener *backtracking* en el lenguaje hace que no siempre sea posible desechar el entorno de una cláusula una vez esta clausula a acabado, pues la presencia de alternativas en cualquiera de los objetivos internos puede hacer necesario volver a esa cláusula y reusar las variables de su entorno.

Pila de control (Control stack): almacena una descripcion de las alternativas pendientes al entrar en una predicado con varias cláusulas, así como el estado de los registros de la máquina. La información relativa a las alternativas se actualiza cada vez que se toma una de ellas. Los *puntos de elección* almacenados en esta pila permiten efectuar *backtracking*, retomando la última alternativa pendiente.

Rastro (Trail): almacena información sobre las ligaduras realizadas a las variables, de forma que se puedan deshacer al volver a una alternativa anterior.

La figura 2.1 es una representación de una WAM completa, con todos sus registros y zonas de memoria.

2.3. &-Prolog

&–Prolog es un lenguaje destinado a expresar paralelismo–And independiente en Prolog. Su sintaxis es la misma de Prolog, salvo por la adición del operador "&"

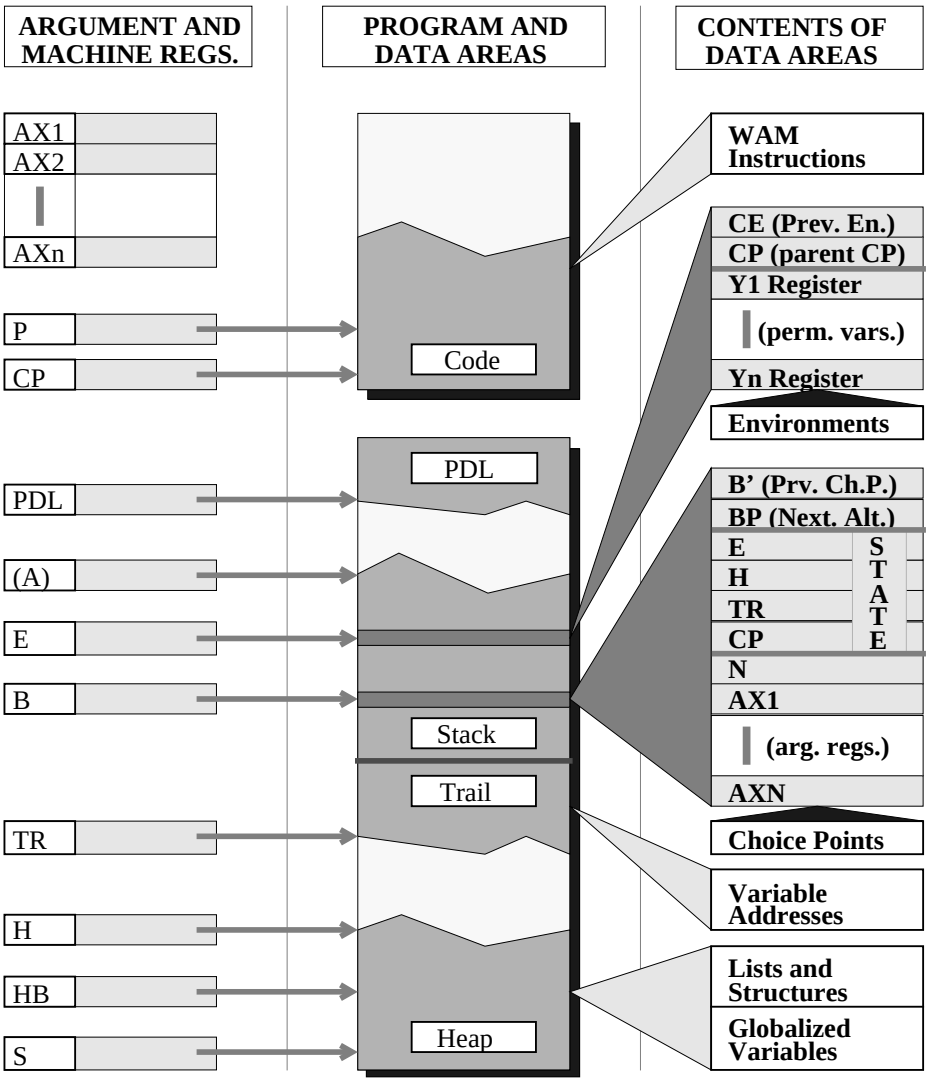

Figura 2.1: Una representación de la WAM

en lugar de "," para expresar ejecución de objetivos en paralelo. Por ejemplo, la cláusula

```
p(X, Y):- a(Z), b(Y)
```

puede reescribirse para su ejecución paralela en &–Prolog como

```
p(X, Y):- (indep(X, Y) ->a(X) & b(Y); a(X), b(y)).
```

y los objetivos `a/1` y `b/1` se ejecutarían en paralelo si son independientes, es decir, si no comparten variables en tiempo de ejecución. La ausencia de variables compartidas permite, desde el punto de vista del programador que escribe una máquina abstracta, que no haya competencia por el acceso a esas variables. La expresión anterior, en la que se usa la construcción *if–then–else* de Prolog junto con predicados para comprobar que se satisfacen las condiciones de independencia, puede escribirse de forma más compacta usando una CGE de la siguiente manera:

```
p(X, Y):- indep(X, Y) =>a(X) & b(Y).
```

Puede haber CGEs en los que la condición de independencia sea vacía. En ese caso puede omitirse la parte condicional, y la cláusula tendrá el siguiente aspecto:

```
p(X, Y):- a(X) & b(Y).
```

Durante la ejecución hacia delante, y en ausencia de fallo, los objetivos que no comparten variables se pueden ejecutar de forma independiente por agentes[1] diferentes, siendo necesaria la sincronización sólo al terminar la CGE, pues la ejecución del resto de la cláusula tras la CGE (denominada *continuación* de la CGE) debe esperar hasta que hayan acabado de ejecutarse todos los objetivos. Cada uno de los objetivos de una CGE da lugar inmediatamente a una tarea por cada activación de a

[1]Denominamos agentes a elementos capaces de ejecutar trabajo de manera independiente unos de otros. La sincronización entre ellos debe expresarse de manera explícita.

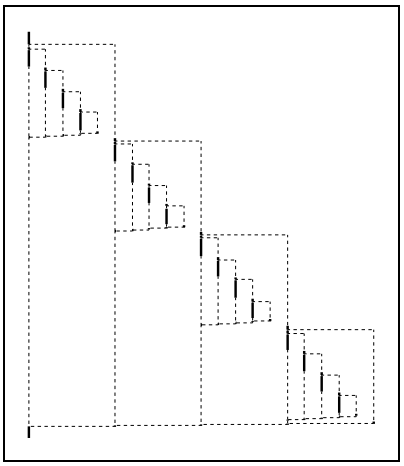

Figura 2.2: Multiplicación de dos matrices, 1 procesador

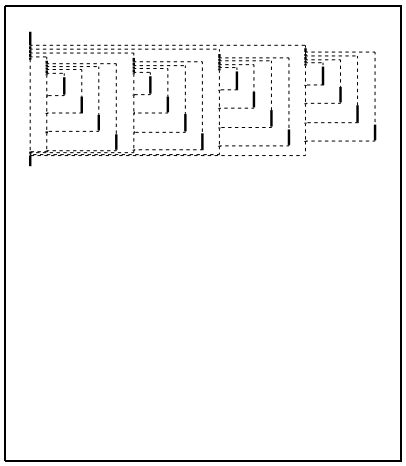

Figura 2.3: Multiplicación de dos matrices, 4 procesadores

cláusula. Asimismo la continuación de una CGE es otra tarea que está cuyo comienzo está sincronizado con la finalización de las tareas en la CGE. Cualquier objetivo en una CGE puede dar lugar a más CGEs, que a su vez podrán producir más tareas.

En ausencia de fallo, el tiempo ideal de ejecución de una CGE es el máximo de la ejecución de todos sus objetivos. En las figuras 2.2 y 2.3 podemos ver la representación gráfica obtenida mediante la herramienta VisAndOr [CGH93] de una ejecución de un programa para multiplicar dos matrices de cuatro elementos.

Los dos gráficos tienen la misma escala de tiempo; el tiempo corre de arriba hacia abajo. Cada uno de los segmentos verticales sólidos representa una tarea secuencial. Los segmentos horizontales punteados son el principio y el fin de CGEs, y lo segmentos verticales punteados son tareas (objetivos paralelos) que están preparadas para la ejecución pero que aún no están siendo ejecutados por ningún agente. Se ve claramente como la ejecución en 4 procesadores lleva menos tiempo, debido a la ejecución de tareas en paralelo. Asimismo se aprecia la estructura de la ejecución del programa: cada CGE espera a que se acaben todas las tareas que empiezan en

él, y tras ello se continúa con la ejecución del resto de la cláusula.

En la ejecución hacia atras (*backtracking*), &–Prolog intenta acercarse lo más posible al comportamiento de Prolog. Podemos distinguir tres tipos de backtracking en &–Prolog:

Local: es el *backtracking* realizado dentro de una tarea secuencial, cuando no es necesaria ninguna comunicación con otras tareas. El mismo mecanismo de Prolog puede usarse directamente.

Interno: ocurre cuando falla alguno de los objetivos de la CGE antes de que la CGE haya acabado. Como Prolog sólo falla en las unificaciones y los objetivos son independientes, las unificaciones que haya hecho cualquiera de ellos no pueden afectar a los otros, de manera que no hay necesidad de relanzar ninguno de los objetivos en la CGE, y se puede retroceder hasta el primer punto de elección anterior a la CGE.

Externo: ocurre cuando falla algún objetivo posterior a la terminación de la CGE. En este caso, para preservar la semántica de Prolog, los objetivos en la CGE que puedan ofrecer alternativas se rearrancan de derecha a izquierda.

2.4. PWAM

El sistema &–Prolog se ejecuta actualmente en un simulador de una máquina abstracta (PWAM [HG91]) construído sobre el sistema SICStus Prolog [Car88] e incluye además un compilador paralelizante [MH89, MH90, WHD88, dlBH92]. El diseño de la PWAM es una adaptación de una WAM secuencial para la ejecución de objetivos paralelos independientes, y es una evolución de la RAP–WAM [Her86a]. La PWAM está diseñada para ser ejecutada en multiprocesadores de memoria compartida. En ella un conjunto de agentes (implementados como procesos UNIX) trabajan

en diferentes PWAMs, realizando objetivos independientes. La memoria compartida permite que todos tengan acceso a las ligaduras de variables que hacen los demás, de forma que cualquier agente puede ejecutar tareas que han sido generadas por otro.

La representación de los datos es la misma que en la WAM. En cuanto a las zonas de memoria, en la RAP–WAM y PWAM existe una nueva pila:

Pila de objetivos: en ella cada agente va almacenando los objetivos que quedan disponibles para su ejecución en paralelo.

Todos los agentes tienen acceso a las pilas de objetivos del sistema (una por agente). La planificación se realiza cogiendo objetivos de las pilas que no estén vacías. Esta estrategia distribuye la planificación y evita que sea un cuello de botella del sistema, debido a que tiene múltiples puntos de entrada. No existe ningún agente "maestro" o encargado de la planificación: todos ellos ejecutan exactamente el mismo código y tienen las mismas responsabilidades.

En RAP–WAM cada objetivo en la pila contenía un puntero al punto de entrada del código del objetivo, un identificador de CGE y del objetivo y una representación de los argumentos del objetivo; se realizaba, por tanto, "goal stacking". Además en las CGE un agente esperaba a que se terminase completamente la CGE para ejecutar la continuación. En la PWAM no se incluyen ya los argumentos del objetivo; en su lugar se almacena con el objetivo un puntero al entorno en el que se había creado la CGE, y el punto de entrada del objetivo se cambia al principio del grupo de instrucciones `put_` que cargan los argumentos. De esa forma se ahorra espacio en la pila de objetivos y se gana velocidad, al realizar la carga de los argumentos en paralelo y al no tener que almacenarlos en la pila. La figura 2.4 representa un momento de la ejecución, en que hay agentes trabajando en PWAMs, PWAMs sin agentes trabajando en ellas y agentes buscando trabajo en el sistema.

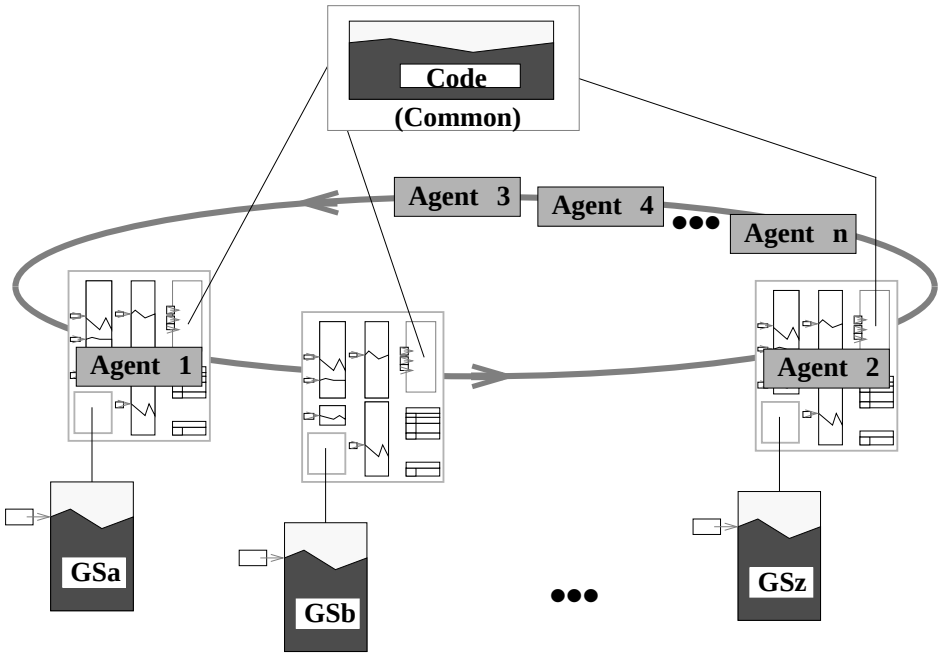

Figura 2.4: Agentes buscando objetivos y PWAMs

varios agentes trabajando en PWAMs y agentes buscando objetivos y PWAMs libres en el sistema.

La compilación de programas &–Prolog a código PWAM necesita de unas pocas intrucciones más para manejar objetivos paralelos y del cambio de algunas otras. No entraremos aquí en la descripción de esas instrucciones. El lector puede referirse a [Her86a] para una descripción de la RAP–WAM y a [HG91] para la PWAM.

Una de las grandes diferencias entre la PWAM y la WAM es la forma en que se usan las zonas de memoria para coordinar y separar las ejecuciones de los distintos objetivos paralelos. Esta coordinación se realiza mediante el empleo de nuevos objetos en las pilas. En particular podemos separar dos clases:

Principios de CGE (Parcall frames): son objetos que residen en la pila local, y extienden un entorno de un cláusula para contener información global acerca