Un Paso en el Proceso de Paralelizacion Automatica de Codigos C

F. Corbera
R. Asenjo
E.L. Zapata

September 2000
Technical Report No: UMA-DAC-00/32

Published in:

XI Jornadas de Paralelismo
Granada, Spain, September 11-13, 2000

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Un paso en el proceso de paralelización automática de códigos C

F. Corbera, R. Asenjo y E. Zapata

Resumen— **Un objetivo aun por alcanzar es la paralelización de códigos que usan estructuras de datos dinámicas. La detección de la estructura de datos usada en el código es uno de los primeros pasos en esta paralelización. En este artículo describimos un método y el compilador que hemos desarrollado para capturar las complejas estructuras de datos generadas, recorridas y modificadas en códigos C. Nuestro método asigna un *Conjunto Reducido de Grafos de forma de Referencias* (Reduced Set of Reference Shape Graphs, RSRSG) a cada sentencia del código para aproximar la forma de las estructuras después de la ejecución de dicha sentencia. Con las propiedades y operaciones que definen el comportamiento de nuestro RSRSG, el método puede detectar con exactitud complejas estructuras de datos recursivas como listas doblemente enlazadas con punteros a árboles cuyas hojas apuntan a otras listas. Se han llevado a cabo otros experimentos con núcleos de códigos reales para validar las prestaciones de nuestro compilador.**

Palabras clave— **Paralelización automática, Estructuras de datos dinámicas, Análisis de forma.**

I. Introducción

La programación paralela es una necesidad en aplicaciones complejas y costosas en tiempo. Los compiladores paralelizadores reducen drásticamente el tiempo necesario para desarrollar un programa paralelo, ya que partiendo de la versión secuencial de un código, generan automáticamente el programa paralelo. Algunos ejemplos de paralelizadores actuales son Polaris, PFA, SUIF, etc, los cuales analizan adecuadamente códigos regulares o numéricos, pero no resuelven el problema con códigos irregulares o simbólicos, basados principalmente en estructuras de datos complejas con punteros. En parte, esto es debido a la gran dificultad implícita en este problema. Sin embargo, debido al uso creciente de estructuras dinámicas en lenguajes de amplia aceptación como C, C++ o Java, éste es un problema que no podemos obviar por más tiempo.

Con esta motivación, nuestro objetivo es proponer e implementar nuevas técnicas de análisis para permitir la paralelización automática de códigos reales basados en estructuras de datos dinámicas. En este trabajo, hemos seleccionado el subproblema de análisis de forma, el cual pretende estimar en tiempo de compilación la forma que las estructuras de datos tomarán en tiempo de ejecución. Con esta información, un análisis posterior podría detectar cuando ciertas secciones del código pueden ser paralelizadas debido a que acceden a regiones independientes de datos.

Departamento de Arquitectura de Computadores, Universidad de Málaga. e-mail: {corbera,asenjo,ezapata}@ac.uma.es.
Este trabajo está soportado por el Ministerio de Educación y Ciencia (CICYT) de España (TIC96-1125-C03), por la Unión Europea (BRITE-EURAM III BE95-1564), por APART: Automatic Performance Analysis: Resources and Tools, EU Esprit IV Working Group No. 29488

Existen varias aproximaciones a este problema. La estrategia más simple [6] consiste en anotar el código secuencial para proporcionar al compilador información extra acerca de la estructura de datos. Por otro lado, otros estudios evitan la intervención del usuario, usando "caminos de acceso" (access paths) o grafos.

En los métodos basados en grafos las "porciones de almacenamiento" son representadas por nodos en el grafo, y las aristas se usan para representar referencias entre ellos. Como ejemplo de estos métodos tenemos los de Plevyak y col. [7] y Sagiv y col. [8]. Estos métodos utilizan un sólo grafo para representar todas las estructuras de memoria. En un trabajo previo [3], combinamos y extendimos ambos métodos para permitir la detección de estructuras más complejas, aunque manteníamos la restricción de un sólo grafo por cada sentencia del código.

En este trabajo, cambiamos ligeramente de dirección, permitiendo la existencia para cada sentencia de varios grafos con más de un nodo sumario.

Entre los primeros estudios que permitían la existencia de varios grafos tenemos los desarrollados por Horwitz y col. [5]. Sin embargo estos métodos son muy inexactos a la hora de analizar estructuras de datos complejas. Un trabajo más reciente en este sentido es el de Sagiv y col. [9] en el cual se representan las estructuras de datos con una serie de predicados tri-valuados. Por lo que nosotros conocemos, los predicados propuestos no son suficientes para analizar las estructuras de datos complejas que manejamos en este trabajo.

El resto del artículo está organizado como sigue. En la sección II describimos brevemente el método completo, introduciendo las ideas clave y presentando un ejemplo de estructura que nos ayudara a entender las propiedades de los nodos. Estas propiedades se describen en la sección III. Hemos implementado este método en un compilador que validamos experimentalmente en la sección IV, mediante el análisis de varios códigos en C basados en estructuras de datos complejas. Por último, concluimos con las principales aportaciones y las propuestas de trabajo futuro en la sección V.

II. Descripción del método

Básicamente, nuestro método aproxima todas las configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia en el código. Podemos llegar a una sentencia siguiendo varios caminos del flujo de control. Cada "camino de control" tiene aso-

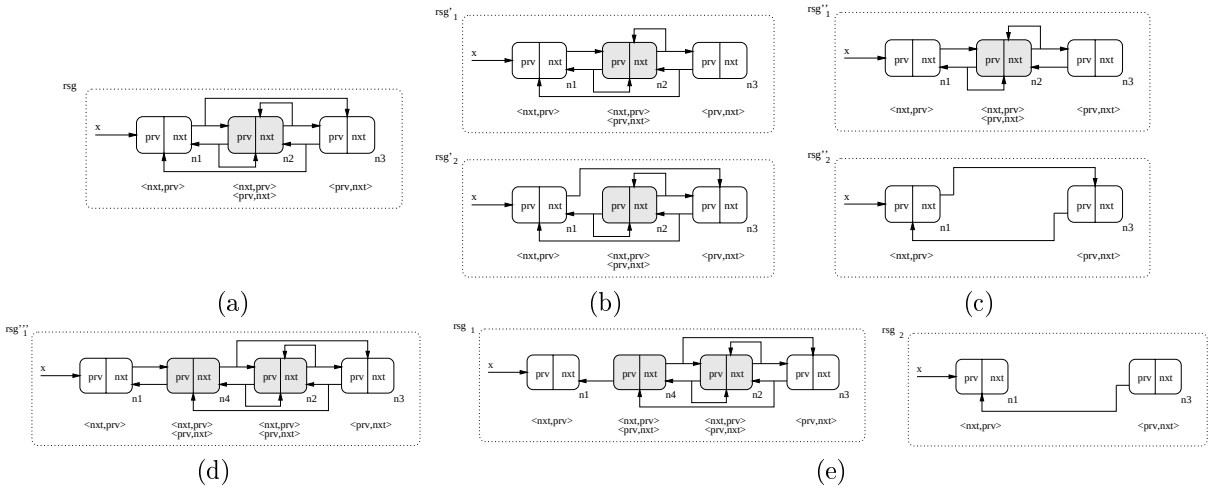

Fig. 1. Ejemplo del proceso completo de la ejecución simbólica de la sentencia $x \rightarrow nxt = NULL$ sobre un RSG.

ciada una configuración de memoria que se modifica por cada sentencia del camino. Nuestro método aproxima cada configuración de memoria por medio de un grafo que denominamos *Grafo de forma de Referencias* (Reference Shape Graph, RSG). Por tanto, cada sentencia del código tendrá asociado un conjunto de RSGs. Este conjunto de RSGs describe la forma de las estructuras de datos existentes después de ejecutar la sentencia.

Este conjunto de grafos lo calculamos mediante la ejecución simbólica del programa sobre los propios grafos. De este modo, cada sentencia transforma los grafos de manera que reflejen los cambios en las configuraciones de memoria representados por los mismos.

Los RSGs son grafos en los que los nodos representan localizaciones de memoria con patrones de referenciado similares. Un único nodo podrá representar varias localizaciones de memoria de forma segura y exacta (si tienen referencias similares) sin perder sus características esenciales.

Cada uno de estos nodos es anotado con una serie de "propiedades" que no son más que las características de las localizaciones representadas por dicho nodo. Por tanto para que dos localizaciones sean representadas por el mismo nodo deben tener las mismas propiedades. Igualmente, dos nodos pueden ser "sumarizados" en un nodo simple si representan localizaciones de memoria de similares propiedades. De este modo, una posible configuración de memoria ilimitada puede ser representada por medio de un RSG de tamaño finito, debido a que el número de nodos diferentes está limitado por el conjunto de propiedades de cada nodo.

Denominamos *Conjunto Reducido de Grafos de forma de Referencias* (Reduced Set of Reference Shape Graphs, RSRSG) al conjunto de RSGs que aproximan todas las posibles configuraciones que puede aparecer después de la ejecución de una sentencia. Es un conjunto reducido, ya que no mantenemos todos los diferentes RSGs que aparecen en cada sentencia. Unimos varios RSGs cuando representan configuraciones de memoria similares. Al igual que para los nodos, para los grafos también hay una serie de propiedades que dos RSGs deben compartir para ser unidos. Por tanto, el número de RSGs distintos asociados con una sentencia también está limitado.

La ejecución simbólica del código consiste en la interpretación abstracta de cada sentencia de manera iterativa hasta que se alcanza un punto fijo en el que el RSRSG asociado con cada sentencia no cambia más [4]. Para cada sentencia que maneja estructuras dinámicas, tenemos que definir la semántica abstracta que describe la modificación producida por la sentencia en el RSRSG. Consideramos sólo seis instrucciones simples con punteros: $x = NULL$, $x = malloc$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, y $x = y \rightarrow sel$. Usando varias de éstas junto con variables temporales se puede construir cualquier sentencia compleja relacionada con punteros. A continuación presentamos de una forma intuitiva las modificaciones producidas por la ejecución simbólica de cada una de las sentencias sobre un RSG:

La sentencia $x = NULL$ elimina las referencias desde la variable puntero (pvar) x a cualquier localización de memoria. De este modo tenemos que eliminar todas esas referencias del RSG.

La sentencia $x = malloc$ simplemente inicializa una nueva localización de memoria, introduciendo en el RSG un nodo nuevo referenciado por x.

La instrucción $x = y$ hace apuntar x al nodo del RSG apuntado por y. Antes de la ejecución de esta sentencia y de la anterior, insertamos automáticamente la sentencia $x = NULL$ para asegurarnos que antes de asignar un nuevo valor a x, dicha pvar no apunta a ningún sitio.

La sentencia $x \rightarrow sel = NULL$ es la más compleja, debido a que rompe enlaces entre nodos lo que produce muchos cambios en las propiedades de los mismos. Para obtener un RSG preciso, antes de borrar el enlace $x \rightarrow sel$, dividimos el RSG en varios rsg_j de manera que en cada uno sólo haya un único destino para $x \rightarrow sel$. A continuación "podamos" cada rsg_j para eliminar los nodos y enlaces que no pertenecen a las configuraciones de memoria representadas por cada rsg_j. Además incrementamos la exactitud del

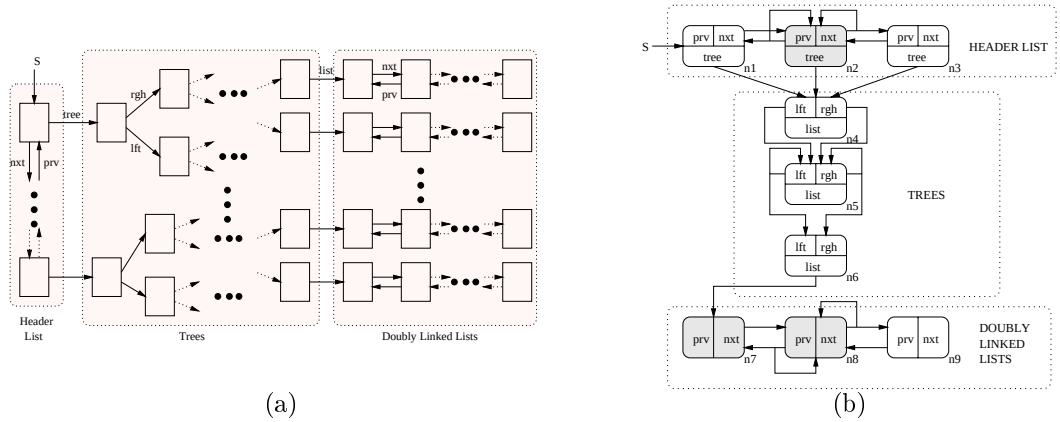

Fig. 2. Una estructura de datos Compleja y el RSRSG resultante.

método, materializando un nuevo nodo de el nodo destino de $x \rightarrow sel$. Después de todo esto, podemos eliminar la referencia de una forma segura. En la Fig. 1 podemos ver un ejemplo gráfico de todo el proceso para ejecutar la sentencia $x \rightarrow nxt = NULL$. En Fig. 1 (a) vemos un RSG que representa una lista doblemente enlazada de dos o más elementos (n_1 representa el primer elemento de la lista, n_2 los centrales y n_3 el último). Los RSGs resultantes de la división se muestran en Fig. 1 (b) (en cada uno de ellos tan sólo existe un destino para $x \rightarrow nxt$). La Fig. 1 (c) muestra el resultado de la poda (donde se han eliminado selectores y nodos que no pertenecen a cada uno de los grafos divididos). Antes de obtener los RSGs finales (tras la eliminación de la referencia $x \rightarrow nxt$) presentados en Fig. 1 (e), vemos que para rsg''_1 es necesaria la materialización de un nuevo nodo n_4 (el único referenciado por $x \rightarrow nxt$) a partir n_2, como puede observarse en Fig. 1 (d). Finalmente eliminamos el enlace $x \rightarrow nxt$ como vemos en Fig. 1 (e).

La instrucción $x \rightarrow sel = y$ implica la ejecución del mismo procedimiento descrito para $x \rightarrow sel = NULL$ (división, poda y materialización) seguido de la generación de un enlace desde el nodo apuntado por x al nodo apuntado por y por el selector sel.

Por último, la sentencia $x = y \rightarrow sel$ introduce nuevas referencias desde x a todas las localizaciones de memoria apuntadas por $y \rightarrow sel$. Ahora, la división, poda y materialización son llevadas a cabo para $y \rightarrow sel$.

Cada rsg modificado por la interpretación semántica, será comprimido sumarizando los nuevos nodos compatibles. Además, varios de estos RSGs serán unidos en un único RSG si representan configuraciones de memoria similares, básicamente si sus relaciones de alias son iguales. Esta operación reduce drásticamente el número de RSGs del RSRSG resultante.

A continuación presentamos un ejemplo de estructura de datos a la cual haremos referencia en las siguientes secciones.

A. *Estructura ejemplo*

La estructura de datos, presentada en la Fig. 2 (a), es una lista doblemente enlazada donde cada elemento apunta a un árbol con el selector *tree*. Además, las hojas de los árboles tienen punteros *list* a otras listas doblemente enlazadas. Distintos árboles no comparten ningún elemento, al igual que las listas apuntadas desde las hojas de los árboles.

Un código en C crea, recorre y eventualmente permuta dos árboles. Nuestro compilador ha analizado dicho código obteniendo un RSRSG para cada sentencia del programa. La figura 2 (b) muestra una representación compacta del RSRSG obtenido para la última sentencia del código. A continuación presentamos la descripción de las propiedades de los RSGs.

III. Grafos de forma de Referencias

Como hemos comentado un RSRSG aproxima todas las posibles configuraciones de memoria tras la ejecución de una determinada sentencia. Este RSRSG esta formado por una serie de RSGs, cada uno de los cuales representa un subconjunto del total de las estructuras representadas por el RSRSG.

Para obtener el RSG que aproxima una configuración de memoria determinada, M, se usa la función abstracta $F : M \rightarrow RSG$. Esta función mapea localizaciones de memoria en nodos y referencias entre localizaciones en referencias entre nodos.

Esta función extrae ciertas propiedades importantes de las localizaciones de memoria y, dependiendo de ellas, dichas localizaciones son trasladadas a nodos. Además si varias localizaciones comparten las mismas propiedades entonces esta función mapea todas en el mismo nodo del RSG. Estas propiedades son: Type, Structure, Simple paths, Reference pattern, Share, y Cycle links.

A. *Type*

Esta propiedad intenta extraer información del texto del código. La idea es que dos punteros de distinto tipo no pueden apuntar a la misma localización de memoria (por ejemplo, un puntero a un grafo y otro a una lista). De esta manera, tomamos la propiedad `TYPE` de una localización, del tipo de la variable puntero usada cuando la localización

es creada. Por tanto, dos localizaciones de memoria pueden ser mapeadas en el mismo nodo si comparten el mismo valor en TYPE.

Esta propiedad provoca que para la estructura de datos presentada en Fig. 2 (a), los nodos que representan localizaciones de listas no sean sumarizados con aquellos nodos que representan localizaciones de árboles, como podemos ver en Fig. 2 (b).

B. Structure

Esta propiedad se usa para evitar la sumarización de nodos del mismo tipo pero que pertenecen a estructuras que no comparten ningún elemento. Es decir, pertenecen a componentes no-conexas. STRUCTURE toma el mismo valor para todas las localizaciones de memoria (y nodos) pertenecientes a la misma componente conexa. De nuevo, dos localizaciones de memoria pueden representarse con el mismo nodo si tienen el mismo valor en STRUCTURE.

C. Simple paths

La propiedad SPATH limita aún más la sumarización de nodos. Llamamos *Simple paths* a caminos de acceso desde variables puntero (pvar) a localizaciones o nodos. Un ejemplo de un *simple path* es $p \rightarrow s$, en el que la pvar p apunta a la localización s. En este ejemplo, el *simple path* para s es $< p >$.

El uso de *simple paths* evita la sumarización de nodos apuntados por las pvars, que son los puntos de acceso a las estructuras de datos. Por tanto, ahora dos localizaciones de memoria se representarán en el mismo nodo si sus *simple paths* coinciden.

D. Reference patterns

Introducimos esta propiedad para representar en diferentes nodos, diferentes localizaciones de memoria con distintos patrones de referenciado. Entendemos por patrones de referenciado (reference patterns) el tipo de los selectores que apuntan a una cierta localización y por los que dicha localización referencia a otras.

Esto es particularmente útil para mantener localizaciones de memoria "singulares" en nodos separados. Por ejemplo, en Fig. 2 (a), la raíz de un árbol es referenciada por la lista cabecera y las hojas no apuntan a elementos del árbol sino a listas doblemente enlazadas. Gracias a los reference patterns, el método produce el RSRSG de Fig. 2 (b), donde la raíz de los árboles, las hojas, y los demás elementos están claramente identificados (n_4, n_6 y n_5 respectivamente).

Para obtener este comportamiento, definimos dos conjuntos SELINset y SELOUTset que contienen el conjunto de selectores de entrada/salida de una cierta localización. En el dominio de los grafos, nuestro método puede llegar a ser incapaz de determinar con exactitud si un selector dado está en el conjunto de selectores de entrada o salida de un nodo n. De este modo, definimos también PosSELINset y PosSELOUTset que contienen selectores que "pueden" apuntar a/desde el nodo n.

Una función booleana C_REFPAT(n_1, n_2) nos indica si dos nodos tienen patrones de referenciado compatibles y por tanto pueden ser sumarizados. Esta función comprueba si los selectores de entrada/salida de los nodos son iguales.

E. Share

Esta propiedad es clave para informar al compilador del paralelismo potencial en una estructura de datos analizada. En particular la información *share* puede decirnos si al menos una de las localizaciones representadas por un nodo es referenciada más de una vez desde otras localizaciones. Es decir, un nodo "shared" representa localizaciones que pueden ser apuntadas desde varios sitios, lo que puede impedir la paralelización de la sección de código que recorre esta localización de memoria.

Debido a la importancia de esta propiedad, usamos dos tipos de atributos para cada nodo:

SHARED(n), es una función booleana que devuelve "true" si alguna de las localizaciones representadas por n es referenciada desde otras localizaciones por medio de diferentes selectores, sel_i y sel_j. Si SHARED(n) es 0, sabemos que si alcanzamos el nodo n por sel_1 y después por sel_2, estamos alcanzando realmente dos localizaciones de memoria diferentes.

SHSEL(n, sel), es otra función booleana que devuelve "true" si alguna de las localizaciones representadas por n puede ser referenciada más de una vez por el selector sel desde otras localizaciones. En un nodo n con un selector sel referenciándose a sí mismo, si SHSEL$(n, sel) = 0$ entonces n no está representando un ciclo en la estructura sino una estructura de tamaño variable.

En la Fig. 2 (b) el nodo central n_2 de la lista cabecera es shared, SHARED(n_2)=1, (los nodos sombreados son shared) debido a que n_2 es referenciado por los selectores nxt y prv. Sin embargo, SHSEL(n_2, nxt)=SHSEL(n_2, prv)=0 lo que significa que siguiendo sólo el selector nxt o prv no es posible alcanzar una localización de memoria visitada anteriormente.

F. Cycle links

El objetivo de esta propiedad es incrementar la exactitud de la representación de la estructura de datos evitando enlaces erróneos que pueden aparecer durante el proceso de modificación de los RSGs. Los "cycle links" de un nodo, n, se definen como el conjunto de pares de selectores $< sel_i, sel_j >$ tal que si partimos del nodo n siguiendo consecutivamente los selectores sel_i y sel_j, se alcanza de nuevo el nodo n.

Por ejemplo, en la estructura de datos presentada en Fig. 1 (a), los elementos intermedios de la lista doblemente enlazada tienen dos cycle links: $< nxt, prv >$ y $< prv, nxt >$, ya que empezando en un elemento de la lista y siguiendo consecutivamente los selectores nxt y prv (o prv y nxt) se alcanza de nuevo el elemento de partida. La propiedad CYCLELINKS se usa durante el proceso de poda de un RSG, para eliminar enlaces que no pertenecen a las

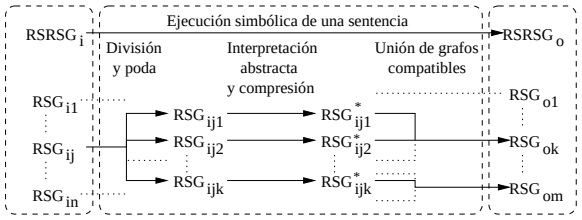

Fig. 3. Descripción esquemática de una ejecución simbólica.

estructuras representadas por dicho RSG.

Una vez vistas todas las propiedades que caracterizan a los nodos, dos nodos de un mismo grafo se sumarizarán en uno sólo si sus propiedades type, structure, shared, shsel y simple paths son iguales, y sus reference pattern son compatibles (`C_REFPAT`). Gracias a esta operación de "compresión", el tamaño de los RSGs está limitado para cualquier estructura.

IV. Resultados experimentales

Todas las operaciones descritas anteriormente han sido implementadas en un compilador escrito en C que analiza código C y genera un RSRSG para cada sentencia del código. Como hemos dicho, la ejecución simbólica de cada sentencia sobre un RSRSG genera un RSRSG modificado. En la Fig. 3 se puede ver esquemáticamente dicha ejecución sobre un $RSRSG_i$ de entrada para generar un $RSRSG_o$ de salida.

Con este compilador hemos analizado el código que genera, recorre y modifica varias estructuras que describimos a continuación.

A. RSRSG de la estructura ejemplo

En la Fig. 2 (b) presentamos la representación compacta del RSRSG resultante en la última sentencia del código que genera, recorre y modifica la estructura de datos presentada en Fig. 2 (a). Vemos que describe con exactitud la estructura de datos en el sentido de que detecta la lista doblemente enlazada acíclica por *nxt* o *prv* y cuyos elementos apuntan a árboles binarios. Como `SHSEL`$(n_4, tree)$=0, podemos decir que dos elementos diferentes de la lista no pueden apuntar al mismo árbol. Al mismo tiempo, dado que ningún nodo de los árboles (n_4, n_5 y n_6) es shared, diferentes árboles no pueden compartir ningún elemento. Lo mismo ocurre para las listas doblemente enlazadas apuntadas desde los árboles: todas las listas son independientes y no hay dos hojas apuntando a la misma lista.

B. Multiplicación dispersa matriz por vector

Presentamos aquí un código irregular que implementa la multiplicación matriz dispersa por vector, $r = M \times v$. La matriz dispersa, M, se almacena en memoria como una lista cabecera doblemente enlazada con punteros a otras listas que almacenan las filas de la matriz. Los vectores, v y r son almacenados también en listas doblemente enlazadas como vemos en Fig. 4 (a).

En Fig. 4 (b) podemos ver la representación compacta del RSRSG obtenido en la última sentencia del código. Como vemos, las tres estructuras implicadas en el código se mantienen en subgrafos separados gracias a la propiedad `STRUCTURE`. En el RSRSG podemos observar que las filas de la matriz son apuntadas desde distintos elementos de la lista cabecera (no hay ningún nodo con la propiedad `SHSEL` "true" para ningún selector). Además, las listas doblemente enlazadas que almacenan las columnas de M y los vectores v y r son acíclicas por los selectores *nxt* y *prv*.

C. Factorización LU dispersa

Esta aplicación resuelve sistemas lineales dispersos no simétricos aplicando la factorización LU de la matriz dispersa, usando un método general [1]. Las columnas de la matriz A se almacenan en listas doblemente enlazadas (ver Fig. 5 (a)), para facilitar la inserción de nuevos elementos y permitir la permutación de columnas.

Después del análisis del código de la LU obtenemos el RSRSG que se muestra en Fig. 5 (b). Como podemos ver, la variable A apunta a una lista doblemente enlazada, la lista cabecera. Cada nodo de esta lista apunta a una lista doblemente enlazada que representa una columna de la matriz. Debido a que `SHSEL` es "false" para todos los nodos y selectores podemos decir que la lista cabecera y las listas columnas son acíclicas cuando son recorridas por un único selector y distintos elementos de la lista cabecera apuntan a diferentes columnas. Un posterior análisis del código y del RSRSG asociado con cada sentencia podría determinar que varias columnas de la matriz dispersa pueden ser actualizadas en paralelo.

D. Simulación N-body Barnes-Hut

Este código se basa en el algoritmo presentado en [2] muy usado en astrofísica. La estructura de datos usada en este código se basa en una representación jerárquica del espacio en un árbol. Por simplicidad hemos usado un árbol binario. En Fig. 6 (a) presentamos una vista esquemática de la estructura de datos usada en el código. La lista apuntada por *Lbodies* almacena los cuerpos del espacio. Cada hoja del árbol apunta a un cuerpo de dicha lista.

La representación compacta del RSRSG en el último paso del algoritmo se presenta en Fig. 6 (b). Vemos que los nodos del árbol no son shared ni por distintos ni por el mismo selector. Un análisis posterior del código puede determinar que podría ser recorrido y actualizado en paralelo.

V. Conclusiones y trabajo futuro

Hemos desarrollado un compilador que puede analizar códigos C para determinar el RSRSG asociado con cada sentencia del código a partir del cual se pueden deducir las propiedades de las estructuras dinámicas de datos. En comparación con trabajos previos, hemos incrementado el número de propiedades lo que nos permite una mayor exactitud en la representación, que es la pieza clave para el análisis del paralelismo exhibido por el código. Hemos vali-

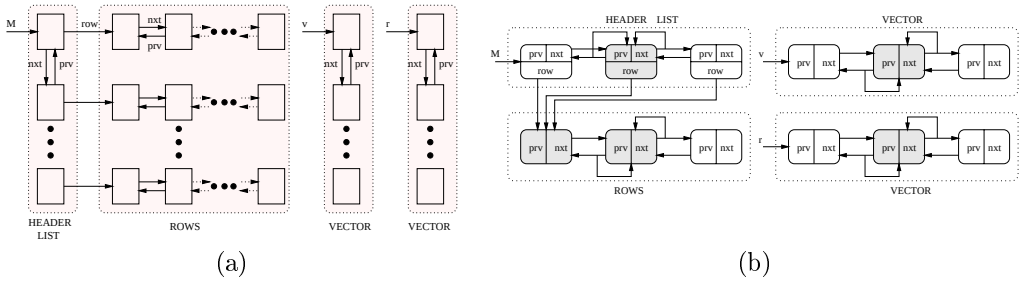

Fig. 4. Estructura de datos y RSRSG compacto para la multiplicación dispersa matriz-vector.

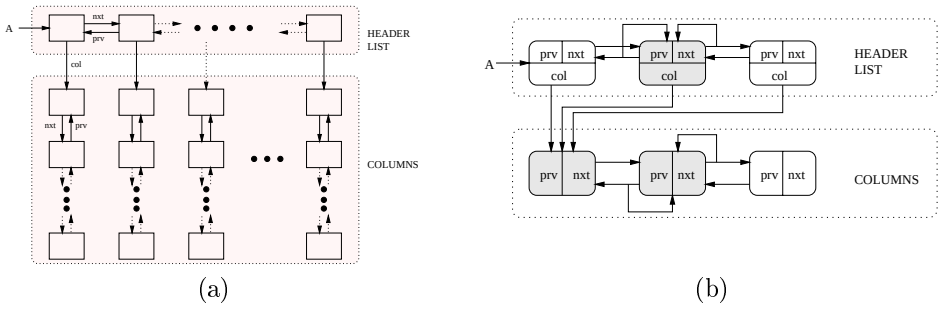

Fig. 5. Estructura de datos y RSRSG compacto para la factorización LU dispersa.

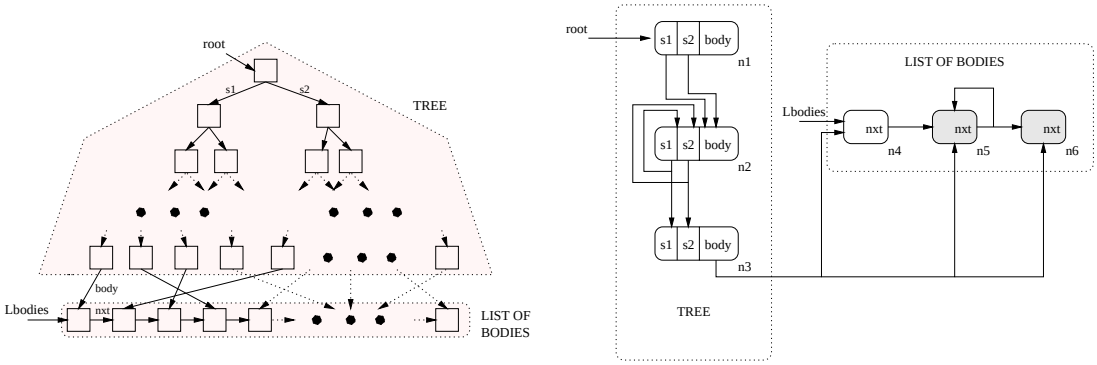

Fig. 6. Estructura de datos y RSRSG compacto del código Barnes-Hut.

dado el compilador analizando varios códigos C que generan, recorren y modifican estructuras dinámicas de datos complejas. Por lo que conocemos, no hay ningún otro compilador que alcance resultados satisfactorios para este tipo de estructuras. Actualmente estamos optimizando la implementación del compilador para reducir el tiempo de compilación y los requerimientos de memoria.

En un futuro próximo desarrollaremos pasos adicionales del compilador que analicen automáticamente los RSRSGs y el código para determinar bucles paralelos en el programa y permitir la generación automática de código paralelo.

Referencias

[1] R. Asenjo. Sparse LU Factorization in Multiprocessors. Ph.D. Dissertation, Dept. Computer Architecture, Univ. of Málaga, Spain, 1997.

[2] J. Barnes and P. Hut. *A Hierarchical O(n· log n) force calculation algorithm.* Nature v.324, December 1986.

[3] F. Corbera, R. Asenjo and E.L. Zapata *New shape analysis for automatic parallelization of C codes.* In ACM International Conference on Supercomputing, 220–227, Rhodes, Greece, June 1999.

[4] P. Cousot and R. Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points.* In Proceedings of the ACM Symposium on Principles of Programming Languages. ACM Press, New York. 238–252, 1977.

[5] S. Horwitz, P. Pfeiffer, and T. Reps. *Dependence Analysis for Pointer Variables.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 28-40, June 1989.

[6] J. Hummel, L. J. Hendren and A. Nicolau *A General Data Dependence Test for Dynamic, Pointer-Based Data Structures.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 218-229. ACM Press, 1994.

[7] J. Plevyak, A. Chien and V. Karamcheti. *Analysis of Dynamic Structures for Efficient Parallel Execution.* In Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds. Lectures Notes in Computer Science, vol 768, 37-57. Berlin Heidelberg New York: Springer-Verlag 1993.

[8] M. Sagiv, T. Reps and R. Wilhelm. *Solving Shape-Analysis problems in Languages with destructive updating.* ACM Transactions on Programming Languages and Systems, 20(1):1-50, January 1998.

[9] M. Sagiv, T. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic.* In Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX, Jan. 20-22, ACM, New York, NY, 1999, pp. 105-118.